Entwicklung eines Deep Learning Verfahrens zur Ladungsrekonstruktion mit dem Übergangsstahlungsdetektor des AMS-Experiments

von

Simon Klüttermann

Bachelor Arbeit in Physik

vorgelegt der

Fakultät für Mathematik, Informatik und Naturwissenschaften

der

RWTH Aachen University

im

September 2018

angefertigt am

I. Physikalischen Institut B

bei

Univ.-Prof. Dr. Stefan Schael
Dr. Henning Gast

Inhaltsverzeichnis

1 Einleitung

Es gibt einen fundamentalen Unterschied zwischen der Funktionsweise eines Computers zu der Funktionsweise eines Gehirns; folglich gibt es Aufgaben, welche ein Computer genauer und schneller erledigen kann (z.B. Rechenoperationen, Speichern von Daten) als ein Mensch, aber auch Aufgaben, welche für einen Menschen trivial sind, während sie für einen Computer tendenziell sehr schwer zu lösen sind. Dazu gehört vor allem die Klassifizierung allgemeiner Daten in spezielle Gruppen: Beispielsweise ist es für einen Menschen leicht einen Apfel zu erkennen, wenn beispielsweise ein Bild einer Frucht vorliegt, während die selbe Aufgabe für einen Computer sehr kompliziert ist; man müsste zuerst versuchen herauszufinden, in welchem Teil des Bildes eine Frucht liegt, dann den Hintergrund herausfiltern und danach einen Index definieren, welcher Äpfel von anderen Früchten unterscheiden kann, aber berücksichtigt, dass sich zwei Äpfel in praktisch jeder ihrer Eigenschaften unterscheiden können (Breite, Höhe, Farbe, Form, Stiel oder kein Stiel).
In den letzten Jahren hat das Wachstum der Rechenleistung moderner Computer es möglich gemacht, die Funktionalität eines Gehirns, bezogen auf ein bestimmtes Problem, sinnvoll zu simulieren. Eine solche Simulation heißt Neuronales Netzwerk. Diese bilden heutzutage eine ganze Industrie, welche sie benutzt um ein breites Spektrum von Problemen, von Wettervorhersagen bis zur Verarbeitung natürlicher Sprache, zu lösen.
In dieser Arbeit soll nun die Möglichkeit der Ladungsbestimmung eines Teilchens durch den Übergangsstrahlungsdetektor des AMS Experiment, mithilfe von Neuronalen Netzwerken, untersucht werden. Während ein klassischer Ansatz sich wohl nur die Höhe der deponierten Energie ansehen würde (vgl. Kapitel 4.2), ist ein Neuronales Netzwerk in der Lage weit mehr mögliche Unterscheidungen zu erkennen. In dieser Arbeit sind, neben der Höhe der Energiedeposition, vor allem Sekundärteilchen von Bedeutung, welche im Übergangsstrahlungsdetektor erzeugt werden. Die Energie dieser ist nach Bethe Bloch Formel[1] proportional zum Quadrat der Ladung des ursprünglichen Teilchens. Aber außer der Fähigkeit, zu lernen, dass deponierte Energie neben der Spur des originalen Teilchens separat von der deponierten Energie auf der Spur zu behandeln sind, wäre ein Neuronales Netzwerk auch in der Lage, die Länge einer solchen Sekundärspur auszuwerten, und nicht korrelierte deponierte Energien zu ignorieren. Der größte Vorteil eines Neuronalen Netzwerks aber liegt darin, automatisch solche Unterscheidungsmöglichkeiten zu erkennen. Leider ist es allerdings so gut wie unmöglich, sich anzusehen, was das ein solches Netzwerk für Unterscheidungen benutzt. Dies ist einer der größten Nachteile beim Einsatz von Neuronalen Netzwerken: Sie können zwar ein gegebenes Problem lösen, aber das sorgt nicht für ein genaueres Verständnis des Problems.
Im Folgenden wird nun eine Einführung in AMS und Neuronale Netzwerke gegeben, danach wird ein Neuronales Netzwerk, welches Ladungen unterscheiden kann, trainiert und vorgestellt.

2 Grundlagen

2.1 Das AMS Experiment

Das AMS[1] Experiment [2] (siehe Abb. 2.1) ist ein Teilchendetektor welcher seit Mai 2011 auf der Internationalen Raumstation installiert ist. Der Vorteil dieses Installationsorts liegt in der Tatsache, dass AMS ohne Störung der Erdatmosphäre kosmische Strahlung vermessen kann. Wissenschaftliche Aufgaben von AMS sind unter anderem die Suche nach Antimaterie und indirekt ein besseres Verständnis dunkler Materie. AMS ist eine Kombination von verschiedenen Subdetektoren, eine kurze Vorstellung dieser folgt:

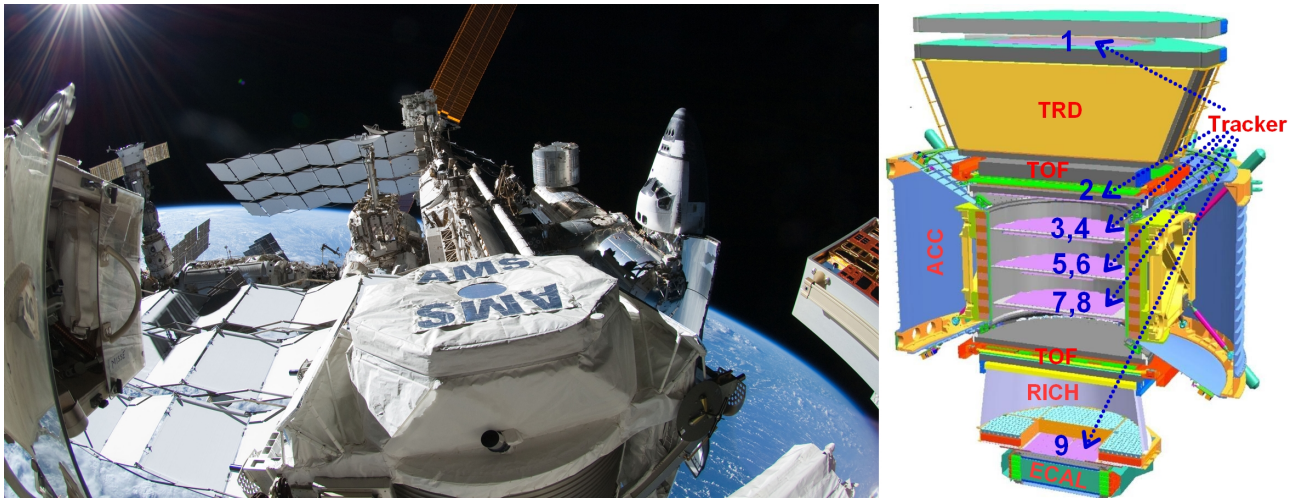

Abbildung 2.1: links: AMS auf der ISS [3], rechts: Aufbau von AMS [4]

Time of Flight
Der Time of Flight (kurz ToF[2]) Detektor, besteht aus vier Szintillatorplatten. Er dient in erster Linie als Trigger für die restlichen Detektoren und um die Flugrichtung, sowie die Fluggeschwindigkeit der Teilchen zu bestimmen. Außerdem ist er in der Lage den Betrag der Ladung der Teilchen (welche proportional zur Wurzel der deponierten Energie ist) vor und nach dem Spurdetektor zu bestimmen.

Spurdetektor
Der Silizium Spurdetektor (Tracker) besteht aus 9 Szintillationsschichten, wovon Schicht 2 bis 8 zwischen einem starken Magnet liegen, während Schicht 1 und Schicht 9 über und unter diesem Magnet liegen. Dieser Aufbau erlaubt es den Pfad eines einfallenden Teilchens zu rekonstruieren, sowie weitere Ladungsmessungen, inklusive des Vorzeichens dieser, durchzuführen. Außerdem lässt sich durch die Ablenkung im Magnetfeld die Rigidität R, also die Proportionalitätskonstante zwischen dem Radius r eines Teilchens in einem Magnetfeld und der inversen Magnetfeldstärke $\frac{1}{\mathrm{B}}$ ($r = R \cdot \frac{1}{B}$, $R = \frac{p}{q}$ mit Impuls p und Ladung q des Teichens) des einfallenden Teilchens bestimmen.

Andere, für die folgende Arbeit weniger wichtige Detektoren beinhalten einen Anti Koinzidenz Zähler (ACC), welcher Teilchen ausschließt, die von der Seite in den Detektor einfallen, einen Tscherenkow Detektor (RICH), welcher die Geschwindigkeit der Teilchen vermisst (zusammen mit der Rigidität und der Ladung lässt sich somit auch die Masse bestimmen und Isotope unterscheiden) und ein elektromagnetisches Kalorimeter (ECAL).

[1] kurz für Alpha Magnet Spektrometer
[2] englisch für Flugzeitdetektor

2.1.1 Der Übergangsstrahlungsdetektor

Der Übergangsstrahlungsdetektor[5] (kurz TRD[3], zu sehen in Abb. 2.2) macht sich die Strahlung zu Nutze, welche entsteht, wenn schnelle Teilchen durch Materialien unterschiedlicher Dielektrizitätskonstanten fliegen. Die auf diese Weise entstehende Strahlung am Übergang solcher Materialien ist recht klein, weshalb dieser Übergang normalerweise oft wiederholt wird. Der TRD welcher am oberen Ende von AMS verbaut ist, erreicht dies durch Vlies Radiatoren (also durch viele zufällige Übergänge, statt periodischen Stapelwiederholungen) welche mit einer Dicke von jeweils 2cm in 20 Schichten angeordnet sind, und deren Übergangsstrahlung (sowie das Signal des ursprünglichen Teilchens) von 5248 jeweils mit Xenon/CO_2 gefüllten Röhrchen, mit 6 mm Durchmesser, vermessen werden. Da ein solches Röhren nur eine Unterscheidung entlang einer Koordinate ermöglicht, sind die ersten und letzten 4 Schichten entlang der X-Achse ausgerichtet, während die mittleren 12 orthogonal dazu entlang der Y-Achse liegen. Die Hauptaufgabe des TRD in AMS ist die Unterscheidung von Teilchen mit vergleichbarer Energie und Ladung aber unterschiedlicher Masse. Das ursprüngliche Problem ist, dass bei großen Energien die Masse nur einen kleinen Teil der Energie des Teilchens ausmacht, weshalb ein Detektor, welcher nur die Energie eines Teilchens, sowie dessen Ladung, misst, nicht mehr zwischen einem Positron und einem Proton unterscheiden kann. Da eine der Aufgaben von AMS die Suche nach Antimaterie ist, muss aber AMS sicher zwischen Positronen und Protonen unterscheiden können. Dies wird vom Übergangsstrahlungsdetektor[4] erreicht, da die abgestrahlte Leistung $I \propto \gamma \cdot q^2$ [6] proportional zum Lorentz Faktor $\gamma = \frac{E}{mc^2}$ ist, also bei gleicher Energie ein circa zweitausend Mal schwereres Proton nur ca ein zwanzigstel eines Prozents der Signale eines Positrons im TRD hinterlässt. In dieser Arbeit ist nun vor Allem die zweite Abhängigkeit, also $I \propto q^2$, sowie die Signale, welche von Sekundärteilchen erzeugt werden, die beim Durchfliegen der Fließ Radiatoren entstanden sind, insbesondere aber auch die Signale, welche direkt beim Durchflug eines kosmischen Teilchens durch ein Röhrchen entstehen, von Bedeutung.

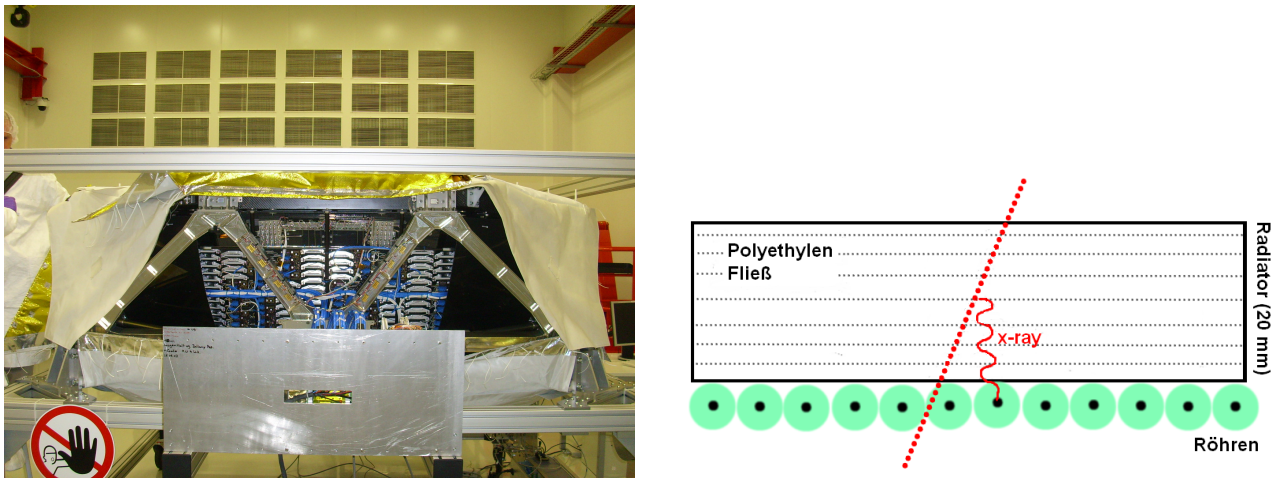

Abbildung 2.2: links: Seitenansicht des Übergangsstrahlungsdetektor vor Einbau in AMS [7], rechts: Funktionsweise des Übergangsstrahlungsdetektors [8]

[3] englisch, Transition Radiation Detektor
[4] und vom elektromagnetischen Kalorimeter

2.2 Neuronale Netzwerke

2.2.1 Funktionsweise neuronaler Netzwerke

Ein neuronales Netzwerk[9] ist im Grunde eine Menge von Matrizen. Diese werden iterativ an einen Eingangsvektor multipliziert werden um einen Ausgabevektor zu erzeugen und eine gesuchte Funktion zu erfüllen. Um die Einträge dieser Matrizen zu finden, existiert eine Verlustfunktion, welche minimal wird für eine gute Anpassung und welche während des sogenannten Trainings nach einer Methode, welche dem Newton Verfahren ähnelt, minimiert wird.
Folglich lassen sich bestimmte anzupassende Funktionen durch bestimmte Größen und Anzahlen der Matrizen ausdrücken. Eine einzelne 1x1 Matrix beispielsweise ist äquivalent zu einer Funktion $f_a(x) = a \cdot x$ mit Parameter a.
Der Nachteil dieses einfachen Modells ist, dass es Funktionen gibt, die sich nicht durch Matrizen darstellen lassen, nämlich solche die nicht linear sind[5]. Dieses Problem wird normalerweise dadurch gelöst, das nach jeder Matrix eine nichtlineare sogenannte Aktivierungsfunktion auf den ausgegebenen Vektor angewendet wird. Dadurch wird die Menge der darstellbaren Funktionen auch auf die nichtlinearen und sogar auf solche Funktionen, welche nicht einmal linear in ihren Parametern sind, erweitert.
Beispielsweise sei hier $f_a(x) = a_1 \cdot x + a_0$ mit einer eindimensionalen Eingabe x und einer typischen Aktivierungsfunktion die einer gespiegelten Fermiverteilung folgt und hier Sigmoid genannt wird ($a(x) = (1 + e^{-x})^{-1}$). Außerdem werden hier wenigstens zwei Matrizen (eine 2x1 Matrix und eine 1x2 Matrix beispielsweise) benötigt, wobei nur nach der ersten Matrix die Aktivierung anwendet wird. Die Idee hier ist, dass die erste Matrix wie eine Identität und eine Konstante agiert, welche von der zweiten Matrix wieder wie ein Problem, welches proportional zur Eingabe ist, verarbeitet werden kann. Dies kann beispielsweise dadurch erreicht werden, das die obere Zeile der ersten Matrix eins ist, während die untere Zeile eine große positive Zahl ist (da die Fermiverteilung für kleine x linear ist, und $\lim_{x \to -\infty} a(x) = 1$ erfüllt). Natürlich werden reale Netzwerke nicht manuell gefüllt, sondern automatisch generiert. Dies geschieht indem in vielen Schritten jeweils das Netzwerk auf den Daten getestet wird, und, mithilfe der oben erwähnten Methode, verbessert wird. Sollte diese Methode konvergieren, wird bewertet wie gut es die gesuchte Aufgabe erfüllt.

Overfitting
Trotzdem ist ein solches Beispiel sinnvoll, da es ein großes Problem Neuronaler Netzwerke zeigt: Das Netzwerk funktioniert nicht mehr wenn die Eingabe null oder gar negativ wird (weil die untere Zeile hier nicht mehr riesige Werte generiert), aber er funktioniert auch nicht mehr wenn zu große Werte eingegeben werden (weil die obere Zeile nicht mehr linear ist). Diese starke Bereichsabhängigkeit ist typisch für Neuronale Netzwerke, heißt aber nicht, dass es nicht möglich ist ein Netzwerk auf anderen Bereichen oder gar auf größeren Bereichen zu trainieren, aber während eine klassische Regression eine Funktion findet welche allgemein gilt, kann man nicht davon ausgehen, dass ein Netzwerk welches man auf einem Bereich trainiert wurde, auch auf einem anderen Bereich funktioniert. Außerdem sieht man auch, dass es nicht unbedingt möglich ist, zu verstehen, wie ein Netzwerk zu seinem Ergebnis kommt.

[5] Das ist nicht ganz genau, auch eine Funktion $f_a(x) = a_1 \cdot x + a_0$ ließe sich nicht durch eine Matrixmultiplikation darstellen, aber man kann das hier umgehen, indem man statt nur einem eindimensionalen Eingangsvektor eine zweite konstante Eingabe verwendet. Diese Methode kann man auch benutzen um andere Abhängigkeiten darzustellen, indem man beispielsweise eine dritte Eingabe x^2 hinzufügt. Der Nachteil dieser Methode ist, dass man Eingangsfunktionen selbst finden muss. Außerdem steigt die Anzahl der Eingangsfunktionen bei mehrdimensionalen Eingabedaten sehr schnell, schlussendlich kann man trotzdem nur Funktionen darstellen kann, welche linear in ihren Parametern sind.

Diese beiden Probleme werden auf die Spitze getrieben in einem Effekt welcher Overfitting [6] genannt wird. Diesen Effekt beschreibt man normalerweise als die Tatsache, dass das Netzwerk nur die Eingabe-Daten auswendig lernt, anstatt sie zu verallgemeinern. In dem oben benutzten Beispiel kann man das so verstehen: wenn es nur zwei Eingangspunkte gäbe um das Netzwerk zu trainieren (also jeweils einen X-Wert und den dazugehörigen Y-Wert $y = f_a(x)$), einen zu $x = 0$ und einen zu $x = 1$, dann reicht das zwar klassisch aus um eine Gerade eindeutig zu bestimmen, trotzdem kann es passieren (insbesondere bei etwas größeren Netzwerken als solchen mit 4 Parametern) dass, das Netzwerk nur lernt zu unterscheiden ob der X-Wert 1 oder 0 ist[7], und je nachdem $f_a(0)$ oder $f_a(1)$ ausgibt. Das mag zwar gute Ausgaben generieren, wenn man das Netzwerk nur 0 und 1 unterscheiden soll, aber im allgemeinen Fall ist ein Netzwerk, welches von beispielsweise $0,5$ bis ∞ denselben Wert ausgibt, nicht zu gebrauchen. Natürlich hat man normalerweise mehr als zwei Eingabe Punkte, aber solange man nicht alle möglichen Eingangsdaten durchgegangen ist (und die Anzahl der möglichen Eingangsdaten steigt exponentiell in der Dimension des Eingangsvektors), ist Overfitting wahrscheinlich der stärkste Effekt, welcher die Qualität der Konvergenz eines Neuronales Netzwerk begrenzt.

Datenteilung

Folglich benötigen man eine Methode um zu testen wie stark ein Netzwerk overfittet. Der einfachste Weg dafür ist es einfach das Netzwerk auf anderen Daten zu testen (genauer: auf Daten, die zwar derselben Struktur folgen wie das originale Datenset[8], aber nicht aus denselben Daten bestehen). In dem gegebenen Beispiel ist das äquivalent zu einem Punkt $0,5$ dessen Ausgabe immer noch einer Gerade folgen soll. Tut er das nicht, und gibt stattdessen beispielsweise $f(0)$ oder $f(1)$ aus, ist das Netzwerk overfittet. In einem realen Trainingsverfahren gibt es ein komplettes zweites Datenset, das sogenannte Validierungsset, auf welchem nach jedem Iterationsschritt dieselbe Verlustfunktion wie auf dem Trainingsset berechnet wird. Ist die Verlustfunktion des Validierungssets deutlich schlechter als die des Trainingssets, ist das Netzwerk overfittet.

Normalerweise folgen beide Verlustfunktionen demselben Verlauf, bis zu einem Punkt, ab welchem die Validierungsverlustfunktion wieder steigt. In diesem Punkt ist das Training fast[9] vorbei, denn weiteres optimieren würde die Ergebnisse wieder schlechter machen. Nun geht es darum das Netzwerk zu bewerten [10]. Dafür wird das Validierungsset benutzt, da das Trainingsset durch Overfitting praktisch beliebig gute Werte produzieren kann. Ein Beispielstrainingsverlauf ist in Abbildung 2.3 zu finden:

[6] englisch für in etwa übermäßige Anpassung

[7] Man könnte sich ein Netzwerk, welches nur Punkte auswendig lernt, als Summe über Produkte von Sigmoids, welche einen Bereich zuschneiden und mit einer Konstanten multipliziert werden vorstellen.

[8] welches im Folgenden Trainingsset genannt wird

[9] Man kann durchaus weiter trainieren, man muss nur die Lernrate senken, vgl. Kapitel 3.2.

[10] vgl. Kapitel 4.1

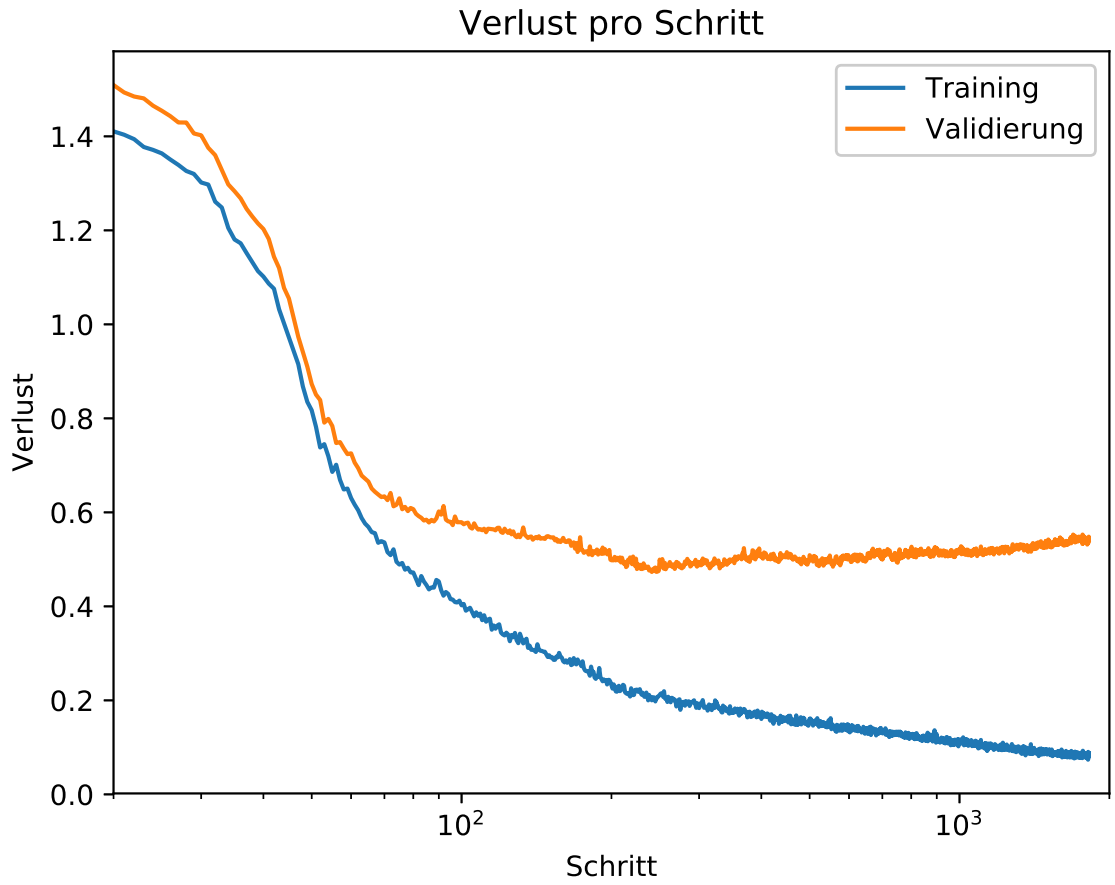

Abbildung 2.3: Beispielverlauf Trainings und Validierungsdaten

Data Leakage

Das produziert aber ein weiteres Problem: Das Validierungsset wird bei und nach jedem neuen Modell benutzt. Das heißt es kann durchaus vorkommen, dass ein auf Validierungsset und Trainingsset optimales Netzwerk nicht nur die Daten des Trainingssets, sondern auch die Daten des Validierungsset, wenigstens teilweise, auswendig gelernt hat. Man nennt diesen Effekt Data-Leakage[11] (In unserem Modell wäre das die Tatsache, dass ein Netzwerk für 0.5 praktisch zufällige Werte ausspuckt, und nur das Modell gewählt wurde, welches den besten Wert ausspuckt). Um diesen Effekt zu testen gibt es normalerweise ein drittes Datenset, das Testset und schlussendlich ist ein Modell nur gut, wenn es auf Validierungsdaten, Trainingsdaten und Testdaten in etwa die selbe (gute) Qualität besitzt, wobei im besten Fall das Testset nur ein einziges Mal benutzt wird, um zu vermeiden, dass das Netzwerk auch die Testdaten auswendig lernt. Im Folgenden beinhaltet das Trainingsset 80% der Daten, das Validierungsset besteht aus 16% der ursprünglichen Daten und das Testset ist 4% groß.[12]

Klassifikation und Regression

Eine wichtige Unterscheidung beim Training eines Neuronales Netzwerk, ist die Frage ob man ein Klassifikations- oder ein Regressionsmodell benutzen möchte. Während Regressionsmodelle wie klassische Regressionen (daher der Name) funktionieren, also eine Menge stetiger Größen

[11] englisch für etwa Daten-Ableitung

[12] Man sollte wohl erwähnen, dass im Folgenden für die finalen Bewertungsindizes trotzdem das Validierungsset benutzt wird, da es viermal so groß ist (und somit die Varianz halbiert) und die Indizes nur mit dem Testset verglichen werden.

auf eine andere Menge stetiger Größen abbildet, ist die Ausgabe von Klassifikationsmodellen quantisiert. Der Vorteil im Vergleich zu Rundung der Werte die ein Regressionsmodell ausgibt, ist die Tatsache, dass man den einzelnen Werten keine Struktur aufzwingt[13], außerdem besteht die Ausgabe des Klassifikationsnetzwerk aus Wahrscheinlichkeiten für jede einzelne mögliche Ausgabe eines zugehörigen Regressionsnetzwerks und nicht nur aus einem gemittelten Wert. Dies funktioniert indem man nicht mehr versucht eine Zahl vorauszusagen, sondern für jede mögliche Ausgabe eine eigene Ausgabevariable definiert, welche entweder 0 oder 1 ist und hier eine weitere finale Aktivierungsfunktion benutzt wird, z.B. eine sogenannte Softmax-Funktion ($\frac{e^{x_i}}{\sum_j e^{x_j}}$), welche die Aufgabe hat, die Ausgabe des Netzwerks auf 1 zu normieren.

Implementation
Zur wirklichen Implementation wird im Folgenden Keras [10] verwendet, welches Tensorflow [11] steuert. In Keras ist ein Neuronales Netzwerk eine Menge von Lagen, wobei jede Matrix, aber auch jede Aktivierungsfunktion eine Lage ist. Die bisher erwähnten Matrizen heißen hier Dense layer (da jede Eingabe einer Matrix eine jede Ausgabe dieser Matrix beeinflussen kann). Es ist aber auch durchaus zu erwähnen, dass es andere Matrizenlagen, insbesondere Konvolutionen, gibt, welche Datenstrukturen besser abbilden können.

2.2.2 Parametrisierte Beschreibung eines Netzwerks

Um ein Neuronales Netzwerk zu beschreiben werden verschiedene Parameter benutzt, welche hier üblicherweise Hyperparameter genannt werden, um sie von den Einträgen der Matrizen zu unterscheiden. Da diese im Folgenden verwendet werden, werden diese hier kurz dargestellt:

Dimension des Netzwerks
Die wohl wichtigsten Hyperparameter eines Neuronales Netzwerk sind die, welche die Größe des Neuronales Netzwerk beschreiben, also die Anzahl der Matrizen und die Größe dieser Matrizen. Je größer desto mehr Matrizen sind, desto kompliziertere Funktionen kann das Neuronale Netzwerk lernen, aber desto wahrscheinlicher wird auch Overfitting.

Feste Funktionen
Um ein Neuronales Netzwerk zu optimieren gibt es außerdem zwei fest einprogrammierte Funktionen, die Aktivierungsfunktion und die Verlustfunktion. Beide müssen vor dem Training ausgewählt werden. Die Aktivierungsfunktion generiert die Alinearität des Netzwerks. Die üblichen Funktionen hier (neben der schon erwähnten softmax Funktion) wirken auf jeden Eintrag eines Vektors separat und sind in Abb 2.4 zu sehen.

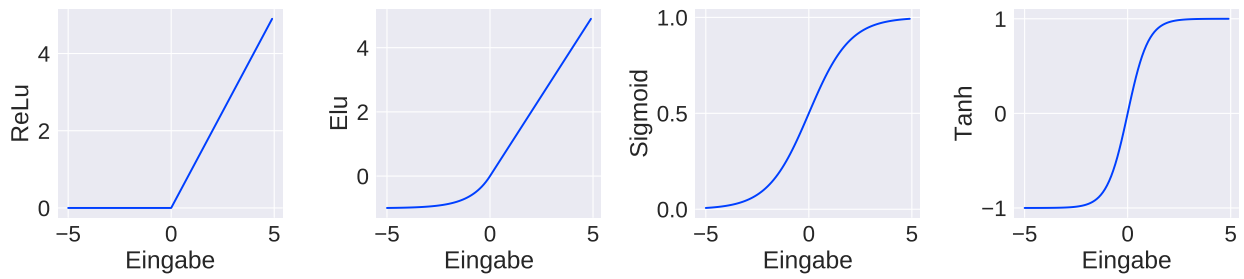

Abbildung 2.4: (1) ReLu: $\Theta(x) \cdot x$, (2) Elu: $\Theta(x) \cdot x + \Theta(-x) \cdot (e^x - 1)$, (3) Sigmoid: $(1 + e^{-x})^{-1}$, (4) Tanh: $\tanh(x)$

[13] Das kann man sich so vorstellen: Ein Netzwerk welches sich nicht sicher ist ob es 5 oder 3 ausgeben soll, gibt bei einem einfachen Regressionsmodell wahrscheinlich einfach 4 aus.

Außerdem gibt es vereinfachte Versionen dieser Funktionen die schneller zu berechnen sind. Die Verlustfunktion wiederum generiert die zu minimierende Größe, welche als $\frac{1}{N} \cdot (\sum_{i=1}^{N} v(Y_{i_{\text{soll}}}, Y_{i_{\text{ausgabe}}}))$ definiert ist, mit der Verlustfunktion $v(x, y)$. Hier muss nun zwischen Klassifikations und Regressionsmodellen unterschieden werden. Während bei Regressionsmodellen eine quadratische Verlustfunktion $f_1(x, y) = (x - y)^2$, eine lineare Verlustfunktion $f_2(x, y) = ||x - y||$ [14] oder eine Mischung beider $f_3(x, y) = \ln(\cosh(x - y))$ [15] üblich sind, ist der Verlust eines Klassifikationsnetzwerk beispielshaft eine Funktion die man Cross-Entropie[16] nennt: $f_c(1, y) = -\ln(y)$, $f_c(0, y) = -\ln(1 - y)$ [17]

Trainings Konstanten

Es gibt außerdem noch zwei Hyperparameter, welche nur den Trainingsverlauf beeinflussen und somit das Newtonverfahren erweitern, welches benutzt wird um das Training durchzuführen. Die erste ist die sogenannte Lernrate, also ein Skalar welcher die Geschwindigkeit der Veränderung in einem Iterationsschritt definiert, wäre die Lernrate null würde das Netzwerk nichts lernen, bei kleinen endlichen Lernraten ist die Konvergenz zwar ziemlich gut, dafür braucht das Training aber bei weitem länger als bei größeren Lernraten. Ist die Lernrate wiederum zu groß kommt es zum Überschwingen, also anstatt einen fast optimalen Wert zu optimieren, werden die Werte so stark verändert, dass sie über das Minimum hinausschießen.
Die zweite Konstante wird Stapelgröße genannt. Anstatt dass der Optimierer (also dass erweiterte Newtonverfahren) eine Funktion aller Eingabedaten ausrechnet und diese minimiert, rechnet er diese Funktion nur von Stapelgröße zufällig gewählten Elementen aus, minimiert diese und fährt mit den nächsten zufälligen Teildatenset fort, bis alle Eingabe-Daten verwendet wurden. Das hat den Vorteil, dass Nebenminima weniger stabil sind (wenn ein Zustand nur bestimmte Daten minimiert, wird er von einem anderen Datenstapel aus diesem Nebenminima gedrängt), dafür kostet es mehr Zeit da mehr Optimierungsschritte benötigt werden, und wenn die Stapelgröße zu klein ist, können einzelne fehlerhafte Daten zu großen Veränderungen in der Anpassung führen, welche bei großer Stapelgröße rausgemittelt würden. Im Allgemeinen macht die Stapelgröße in Bereichen, welche die Zeit nicht deutlich erhöhen, keinen großen Unterschied, bis zu einer Grenze, ab welcher die Indizes schnell deutlich schlechter werden, folglich möchte man am besten eine Stapelgröße verwenden, welche kurz vor dieser Grenze liegt.
Außerdem gibt es eine große Menge verschiedener Optimierer, welche alle deutlich besser optimiert sind um Netzwerke zu trainieren, als die hier vorgestellte Erweiterung eines einfaches Newtonverfahren, aber immer noch ähnlich funktionieren. Im Folgenden wird Adam [12] für Regressionsprobleme und RMSprop [13] für Klassifikationsprobleme benutzt.

Regulierungsmethoden

Um Overfitting zu verhindern gibt es ebenfalls verschiedene Methoden. Eine nennt man Regulierung. Die Idee hier ist, das Netzwerk für große Beträge von Matrixeinträgen zu bestrafen [18]. Die Höhe der Strafe ist die Regulierungskonstante[19]. Dies kann man verstehen wenn man sich eine Polynomregression an eine Funktion wie x^2 mit einem gewissen Fehler ansieht, deren Regression riesige Zahlen mit wechselndem Vorzeichen vor hohen Potenzen ergibt, welche die

[14] Der Unterschied zwischen linearen und quadratischen Verlustfunktion äußert sich darin, dass es bei einem quadratischen Fehler weniger wichtig ist, ob ein Wert leicht abweicht, während er große Abweichungen stärker bestraft.

[15] Nach Taylorentwicklung verhält sich $\ln(\cosh(x))$ für kleine x wie $\frac{x^2}{2}$ und für große x linear, er bestraft also weder kleine Fehler noch große Fehler zusätzlich.

[16] englisch für etwa Kreuz-Entropie

[17] man beachte dass es hier reicht $f_c(x; y)$ für $x = 1$ und $x = 0$ zu definieren, da die Sollwerte eines Klassifikationsnetzwerks nur diese Werte annehmen können

[18] Bestrafen heißt hier die zu minimierende Funktion zu erhöhen.

[19] Man könnte auch noch verschiedene Regulierungsmethoden (linear, quadratisch) unterscheiden.

Fehler auf die Funktion beschreiben ohne die Parabel zu erkennen.
Eine andere Methode heißt Dropout[20]. Die Idee hier ist es, bei jedem Trainingsschritt einen zufälligen Teil des letzten Ausgabevektors auf null zu setzen[21]. Dieser Anteil ist die gesuchte Konstante (im Folgenden einfach Dropout genannt). Dies hat zum einen den Effekt, dass das Netzwerk mehrfach dasselbe lernen muss, und am Ende diese Werte mittelt. Dies hindert das Netzwerk daran, komplett in einem Nebenminima zu verweilen, aber auch, dass das Ergebnis unabhängig von einzelnen Parametern immer noch akzeptabel sein muss. Dies bedeutet in dem oberen Beispiel, dass, wenn eine Konstante (welche den Fehler beschreibt[22]) in der Polynomregression wegfällt, sich die Qualität der Regression nicht sonderlich stark ändern darf. Dies ist nur gegeben, wenn die Konstante klein ist.

[20] englisch für etwa Ausfall

[21] und die restlichen Werte mit einer Konstante zu multiplizieren, um die Höhe der Ausgabewerte konstant zu halten

[22] da die eigentliche Funktion in einem Neuronalen Netzwerk mit Dropout durch mehrere äquivalente Konstanten beschrieben wird, es also weit unwahrscheinlicher ist, dass sie komplett wegfällt

3 Training

Das grundlegende Ziel der vorliegenden Arbeit ist es, ein Neuronales Netzwerk zu trainieren, welches die Rohdaten des Übergangsstrahlungsdetektors ausließt, und aus diesen die Ladung eines einfallenden Teilchens bestimmt. Dies kann zuerst einmal vor Allem dadurch funktionieren, dass die deponierte Energie im Übergangsstrahlungsdetektor proportional zum Quadrat der Ladung des Teilchens ist [23], aber es kann auch noch deutlich besser werden, da in den Rohdaten mehr Information über die Ladung steckt, als nur in der gesamten deponierten Energie[24] Der Vorteil bezogen auf AMS ist die Tatsache, dass der Übergangsstrahlungsdetektor hier recht früh im Verlauf des Teilchens sitzt, somit könnten durch eine Ladungsbestimmung im Übergangsstrahlungsdetektor und eine weitere in einem späteren Detektor, Teilchen herausgefiltert werden, welche in AMS wechselgewirkt haben.

3.1 Ereignisselektion

3.1.1 Datengeneration

Um ein solches Netzwerk zu trainieren, benötigt man Eingangsdaten und dazugehörige Ausgabedaten. Außerdem wird für bessere Vorhersagen auch die Rigidität des Teilchens benötigt. Als Eingabedaten werden für jede der 20 Lagen des TRD, die deponierte Energie in 50 Bins um die rekonstruierte Spur herum verwendet. Hierbei ist ein Bin 6 mm breit ist, entspricht also einem Röhrchen. Außerdem wird nicht zwischen der Orientierung der Röhren unterschieden. Schlussendlich ergibt das 1000 Datenpunkte im Ausgabeformats eines Analog zu Digital Umsetzers (also ADC[25] Counts). Beispiele sind in Abb. 3.1 zu sehen.

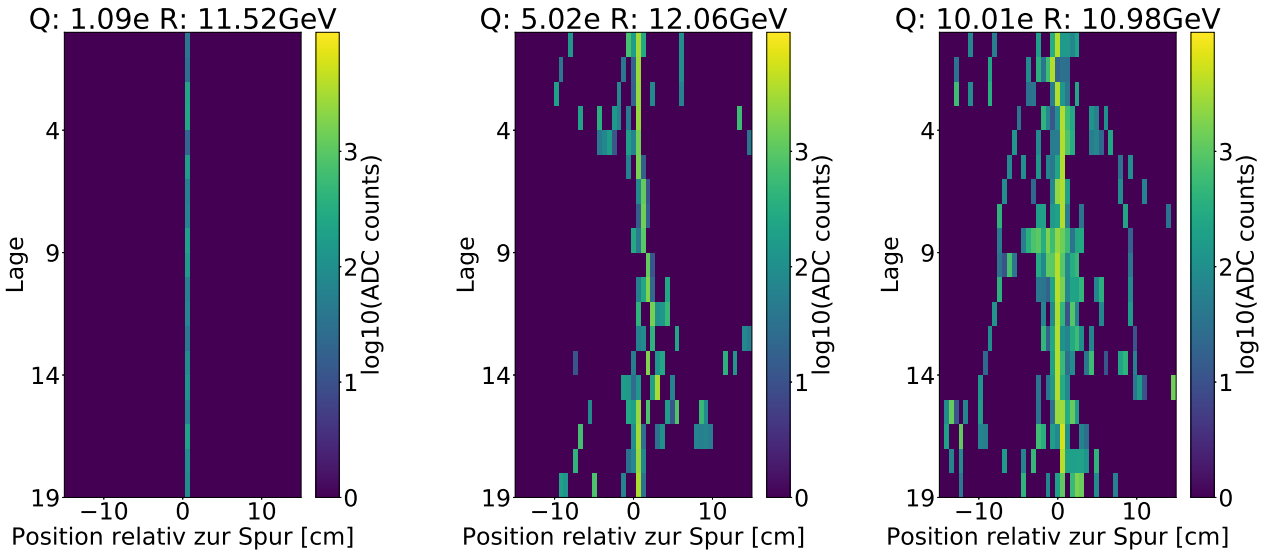

Abbildung 3.1: Beispiel Rohdaten mit Ladung (1): 1, (2): 5, (3): 10

Als Ausgabedaten wird die Ladung aus den Ladungen, welche das ToF und die Lagen des Spurdetekors ausgeben, gewichtet gemittelt. Die dafür benötigten Fehler werden aus den Breiten der Verteilung der zugehörigen Ladung bestimmt. [26]

[23] vgl. Kapitel 2.1.1
[24] vgl. beispielsweise Anhang IV, insbesondere Bild IV.4
[25] Analog Digital Converter
[26] vgl. Anhang IV

Optimiertes Eingabeformat
Während des Optimierens hat sich ergeben, dass eine Liste direkter ADC Counts nicht die bestmögliche Eingabe sind[27] und dass das ein deutlich besseres Eingangsformat neben allen Datenpunkten welche auf der Spur und direkt neben dieser liegen, nur noch Summen über alle Lagen eines jeden Bin enthält. Da solche Netzwerke nicht nur qualitativ besser sind, sondern auch schneller agieren, werden im folgenden nur Netzwerke, welche auf zugeschnittenen Daten agieren, vorgestellt.

3.1.2 Vorselektion

Daten werden generiert nach dem AMS Datenset B950 pass6 zwischen dem 20 Mai 2011 und dem 12 November 2017 durch die Software ACQt 7.6 Da schon wenige schlechte Eingabedaten Overfitting stark begünstigen, ist die Selektion der Daten besonders wichtig. Hier heißt das (Der Verlust ist jeweils nur dieser in Events, welche alle vorherigen Cuts bestanden haben):

Name des Cuts	Verlust für kleine Ladungen	Verlust für große Ladungen
keine Fehlerhaften Ereignisse	4,82%	6,13%
Wenigstens ein Teilchen	0,05%	0,05%
Besitzt eine Spur im Spurdetektor	41,94%	41,41%
positive Rigidität	18,62%	18,42%
Hat eine ToF Ladung	0%	0%
ToF Ladung zwischen 0 und 2.5	4,21%	-
ToF Ladung größer 2.5	-	95,59%
mindestens 7 Hits im Trd	1,58%	1,7%

Tabelle 3.1: Verlust pro Cut in der Vorselektion

Wobei von den Ladungscuts jeweils nur einer angewendet wird, da es bei weitem mehr Ereignisse mit kleiner Ladung gibt. Hier wird somit die Datengeneration auf 2 verschiedenen Datensets ausgeführt: Auf 70 zufälligen Tagen[28] für kleine Ladungen und auf allen Daten (zur Zeit der Datengeneration sind das 2304 Tage) für große Ladungen. Insgesamt bleiben somit 1270 Millionen Ereignisse für kleine Ladungen und 2030 Millionen Ereignisse für große Ladungen übrig.

3.1.3 Nachselektion

In einer weiteren Filterstufe werden folgende Filter die Ladung angewandt[29]:

- beide ToF Ladungen weichen nicht mehr als 0,3 von der Inneren Spurdetektorladung ab
- jede Ladung größer als 0.3, welche keine Fehler hervorruft fließt in den gewichteten Mittelwert ein
- wenigstens 2 Ladungen (von 5) fließen in den gewichteten Mittelwert ein
- Der Wert der gemittelten Ladung weicht nicht mehr als 0,3 von dem gerundeten Wert dieser ab

Diese Selektion ist recht streng, aber eine Abschwächung dieser führt zu einer deutlich schlechteren Anpassung. Dies ist valide, da die Cuts jeweils nur auf die benutzten Ausgabewerte wirken, und somit, mit Ausnahme des ersten Cuts[30], nicht mit den Eingabewerten korrelieren sollten.

[27] vgl. Anhang III
[28] siehe Tabelle VI.3
[29] diese Filter sind das Ergebnis einer Optimierung, wenn diese auch weit weniger genau ist, als Beispielsweise die der Hyperparameter
[30] Der erste Cut sichert ab, dass Teilchen nicht in AMS so wechselgewirkt hat, dass das Teilchen seine Ladung wechselt

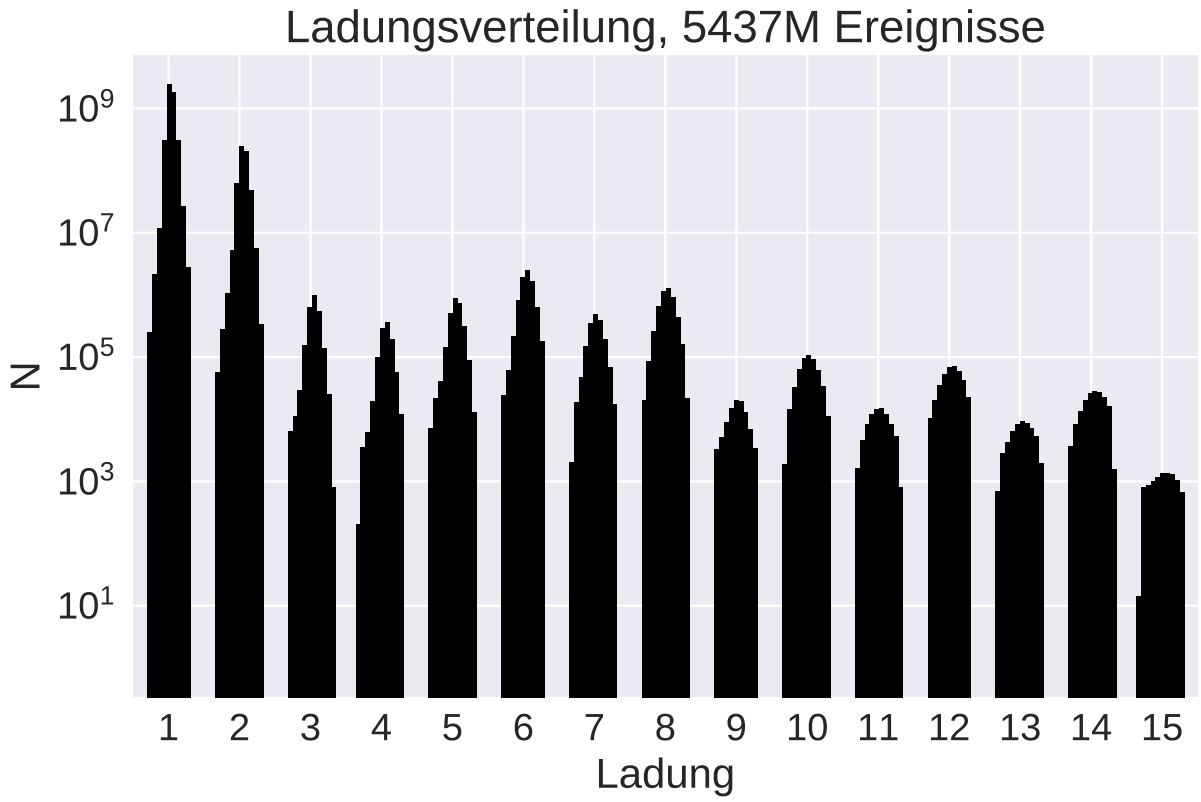

Abbildung 3.2: Ladungsverteilung nach der Nachselektion, Ladungsschnitt herausgerechnet

Die schlussendlich erhaltenen Daten sind in Abb. 3.2 zu sehen und werden für jede Ladung von 1 bis 8 normiert[31], in Trainings, Validierungs und Testdaten aufgeteilt[32] sowie so zusammengefügt, das von jeder Ladung gleichviele Ereignisse vorhanden sind und schlussendlich wird die so erzeugte Liste von Ereignisse gemischt. Jetzt liegt die Anzahl der Trainings Daten bei 6,4 Millionen Ereignissen, sowie die Anzahl der Validierungs und Test Daten bei 1,28 Millionen und 0,32 Millionen.

3.1.4 Modelcharakteristika

Die in Tabellen 3.2 und 3.3 angegebenen Modelle sind das Ergebnis extensiver Optimierungen, mehr dazu in Anhang II und III

[31] ein Listenelement wird durch den Mittelwert dieses Elements auf allen Daten geteilt, bei denen dieses Element von Null verschieden ist. Dies soll sicherstellen, dass alle verwendeten Anregungen die gleiche Größenordnung haben, auch wenn sie nur sehr selten angeregt werden

[32] vgl. Kapitel 2.2.1

Regression

Parameter	Wert
anfängliche Lernrate	$5 \cdot 10^{-4}$
Stapelgröße	500
Aktivierungsfunktion	vereinfachter Sigmoid
Netzwerk Dimensionen	300, 20, 8
Dropout	7% nach dem ersten Layer, kein Dropout sonst
Verlustfunktion	gemittleter absoluter Fehler

Tabelle 3.2: Hyperparameter des Regressionsmodells

Klassifikation

Parameter	Wert
anfängliche Lernrate	$5 \cdot 10^{-4}$
Stapelgröße	100
Aktivierungsfunktion	Elu, sowie ein Softmax nach dem letzten Layer
Netzwerk Dimensionen	300, 15
Dropout	30% nach dem ersten Layer, kein Dropout sonst
Verlustfunktion	Kategorische Crossentropie

Tabelle 3.3: Hyperparameter des Klassifikationsmodells

3.2 Trainingsverlauf

Um optimale Konvergenz zu erreichen, trainiert ein Netzwerk solange, bis es 10 Schritte[33] lang die zu optimierende Größe (also die mittlere absolute Abweichung für Regressionsnetzwerke und die Wahrscheinlichkeit Richtig zu raten für Klassifikationsnetzwerke) nicht verbessert hat, dann wird die Lernrate mit einen Faktor 0.3 verkleinert. Sollte das irgendwann nicht mehr helfen, die zu optimierende Größe zu verbessern, bricht das Netzwerk nach insgesamt 25 Steps ohne Fortschritt das Training ab und das Modell wird benutzt, welches die beste zu optimierende Größe auf Validierungsdaten vorweist. Da ein solcher Trainings Vorgang recht lange dauert (Größenordnung Stunden), gibt es auch ein alternatives kleineres Datenset (mit ca. 100 Tausend Ereignissen, Zeitdauer Größenordnung Minuten), auf dem Modellparameter optimiert werden [34], welches aber natürlich deutlich schlechtere Ergebnisse liefert. Beispiele eines solchen Trainings sind in Abbildung 3.3 zu finden.

[33] ein Schritt ist eine Wiederholung aller Daten

[34] vgl. Anhang II und III, insbesondere III für eine Begründung der Optimierung auf kleinen Datensets

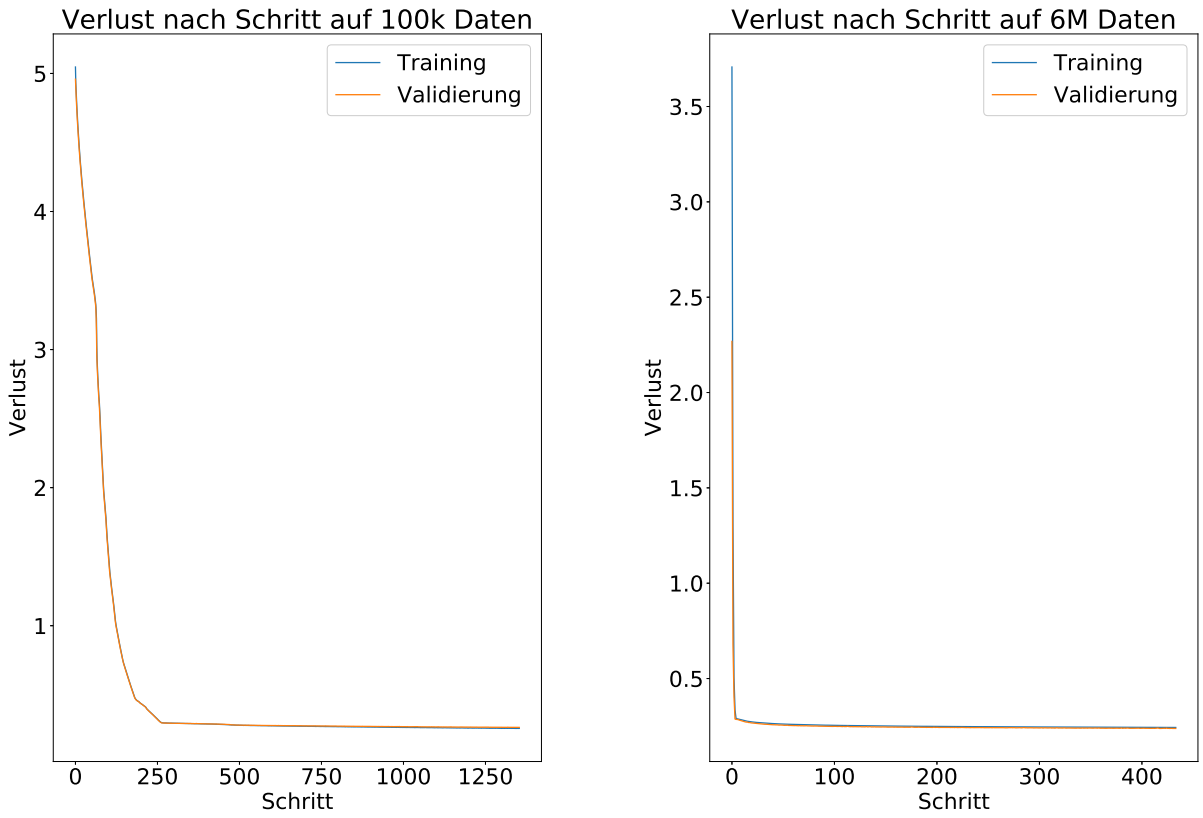

Abbildung 3.3: Ein Beispieltrainingsverlauf, links auf 100k Ereignissen, rechts auf 6M Ereignissen

4 Analyse

4.1 Bewertungsmethoden

Um ein Netzwerk zu bewerten, werden verschiedene Methoden verwendet, eine Auflistung dieser folgt.

Abweichung
Der wohl einfachste Weg ein Neuronales Netzwerk zu bewerten ist es, einfach den mittleren Abstand zwischen dem gewollten und dem erreichten Wert zu berechnen, folglich $\Delta = \sum_i ||Y_i - N(X_i)||$. Der Vorteil dieser Methode ist, dass sie dem entspricht, was ein typisches Regressionsmodell während des Trainings minimiert. Das heißt Qualitäten lassen sich leicht vergleichen, auch wenn das Netzwerk noch nicht komplett konvergiert ist. Außerdem entspricht dieser Index gut bestimmten Erwartungen (so etwas wie der Index wird größer für extreme Stapelgrößen oder Lernraten, siehe dazu auch Anhang II.1)

Wahrscheinlichkeit
Ein anderer Index ist die Wahrscheinlichkeit dafür, dass die gerundete Soll-Ausgabe nicht der gerundeten realen Ausgabe entspricht (p). Der größte Unterschied zu Δ ist, dass dieser Index kleine (und sehr große) Schwankungen in der Ausgabe praktisch ignoriert. Das heißt, mit p kann man testen, ob Ausgabewerte wirklich häufiger richtig werden oder ob nur deren Verteilung schmaler wird. Ein weiterer Vorteil von p ist die Tatsache, dass p quantisiert ist, das heißt aus einer Verbesserung von p kann man herausrechnen wie viele Elemente sich verändert haben und ob das nur Zufall oder eine wirkliche Verbesserung ist (da deren Fehler durch die Poisson Verteilung gegeben ist: $\mathrm{p} = \frac{N_{\mathrm{falsch}}}{N_{\mathrm{total}}}$, $\sigma_{\mathrm{p}} = \frac{\sqrt{N_{\mathrm{falsch}}}}{N_{\mathrm{total}}}$). Außerdem zeigt p eine hohe Korrelation zu Δ, wenigstens in Hyperparametervariationen.

Anpassungsgüte
Außerdem kann man auch einfach das $\frac{\chi^2}{ddof}$ ausrechnen (Dabei sollte man erwähnen, dass die Anzahl der Parameter des Neuronalen Netzwerks im Folgenden vernachlässigt werden, dies ist nötig, da die Anzahl der Parameter die Anzahl der Validationsdaten durchaus übersteigen kann [35]. Dies ist aber auch gerechtfertigt, weil die Daten auf denen Indizes berechnet werden, nicht dieselben sind die für das Training benutzt werden, trotzdem sollte man χ wohl eher als ganz normalen Index sehen, welcher nur den Namen χ trägt). Also $\chi = \sum_i (\frac{Y_i - N(X_i)}{\sigma_i})^2$. Der Vorteil hier ist, dass Werte mit ihren Fehlern gewichtet werden, was fehlerhafte Sollwerte filtert. Da aber die Fehler relativ konstant sind, bringt dies nicht sonderlich viel. Außerdem werden große Abweichungen stärker gewichtet und kleine weniger stark, als in Δ, das heißt man kann χ als eine Kombination von Δ und p sehen.

Zeit
Der letzte Index ist die Zeit, welche ein Netzwerk braucht um ein Ereignis zu verarbeiten. Dies mag zwar nicht viel über die Konvergenzqualität aussagen, trotzdem werden diese Werte im Folgenden immer angegeben, da sie durchaus stark schwanken können und für die Implementation wichtig sind

[35] Dies mag zwar seltsam erscheinen, ist aber (vor allem mit Dropout) kein Grund dafür diese Anzahl zu senken.

4.2 Ein Modell ohne Neuronale Netzwerke

Um zu testen wie nützlich neuronale Netzwerke sind, um Ladungen aus dem TRD vorherzusagen, hier eine recht einfache Lösung ohne Neuronales Netzwerk: Die deponierte Energie pro Strecke ist nach Bethe Bloch Formel[1] abhängig von der Ladung $\frac{dE}{dx} \propto q^2$. Wenn man nun die Anzahl der Teilchen, welche im TRD absorbiert werden vernachlässigt, sollte die gesamte deponierte Energie im TRD proportional zur $\sqrt{q}$ sein. Wenn man aber über alle Bins summiert und für jede Ladung (von 1 bis 8) über alle Ereignisse mit dieser Ladung im Trainingsset mittelt, generiert man zusätzlich zu einer Wurzel noch einen Anstieg bei hohen Energiedepositionen (Also Sättigungseffekte), folglich wird an diese Punkte keine Wurzel angepasst, sondern ein Polynom 7ten Grades, siehe dazu Abbildung 4.1. Da ein solches Polynom nach dem höchsten verwendeten Punkt stark steigt, wird jede Ausgabe, welche 8 überschreitet auf 8 gesetzt.

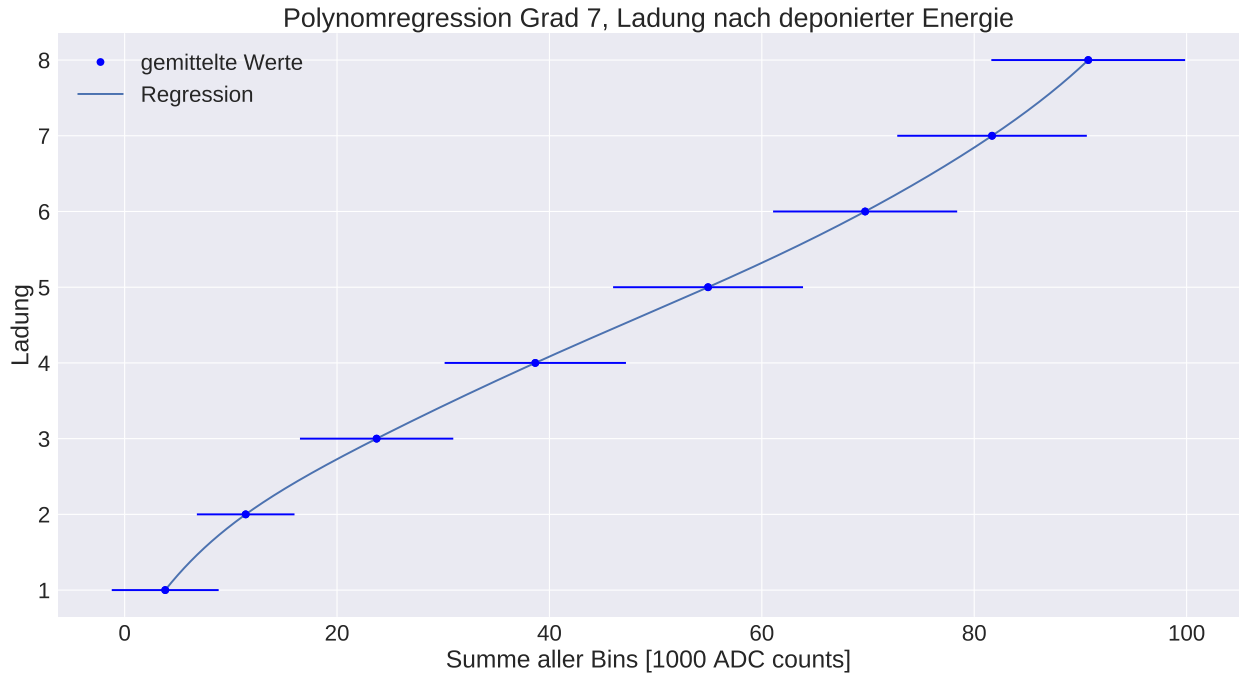

Abbildung 4.1: Polinomregression Ladung als Funktion der Deponierten Energie, Fehler hier die Schwankung eines einzelnen Ereignis

Bewertung

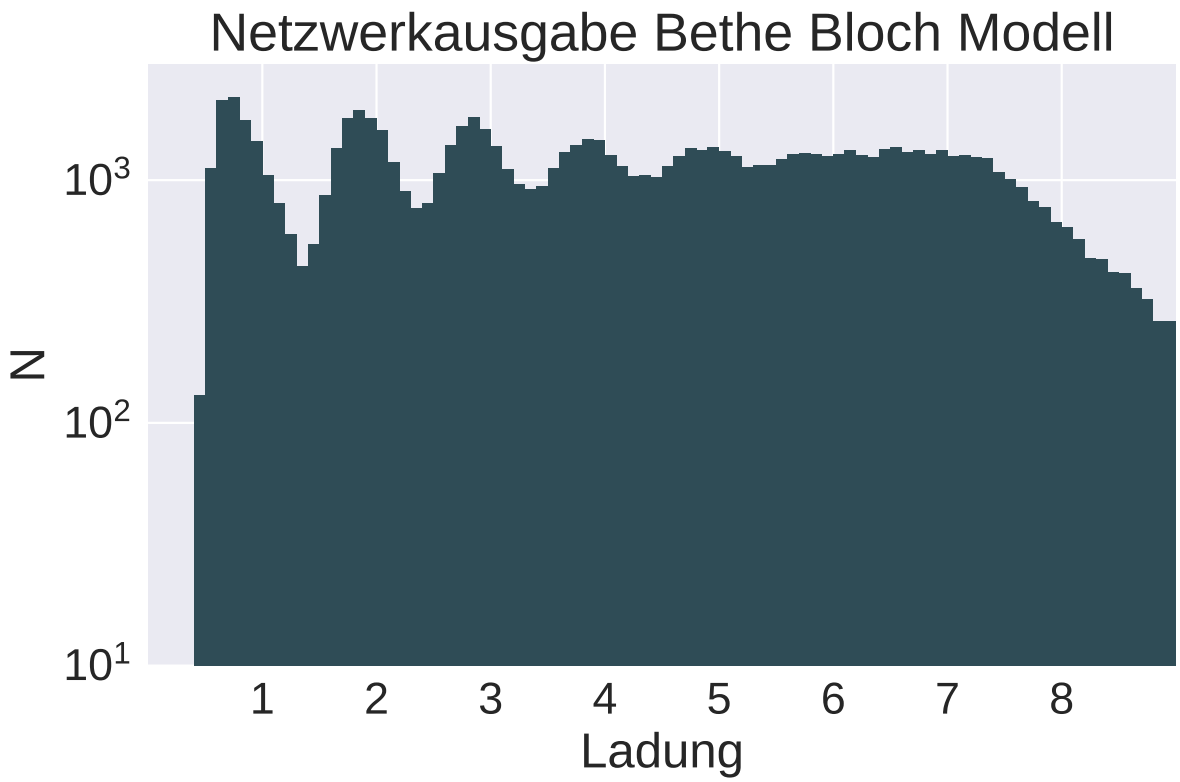

Abbildung 4.2: Ausgabeverteilung der Ladung welche nach Bethe Bloch Modell vorhergesagt wurde

Wenn man sich die Fehler der Regressionspunkte (siehe Abb. 4.1), oder die Peakbreite der Ladungsverteilung (siehe Abb. 4.2), ansieht, zeigt sich, dass dies nicht die beste Methode ist, um die Ladung eines Teilchens zu bestimmen.

4.3 Das Regressionsmodell

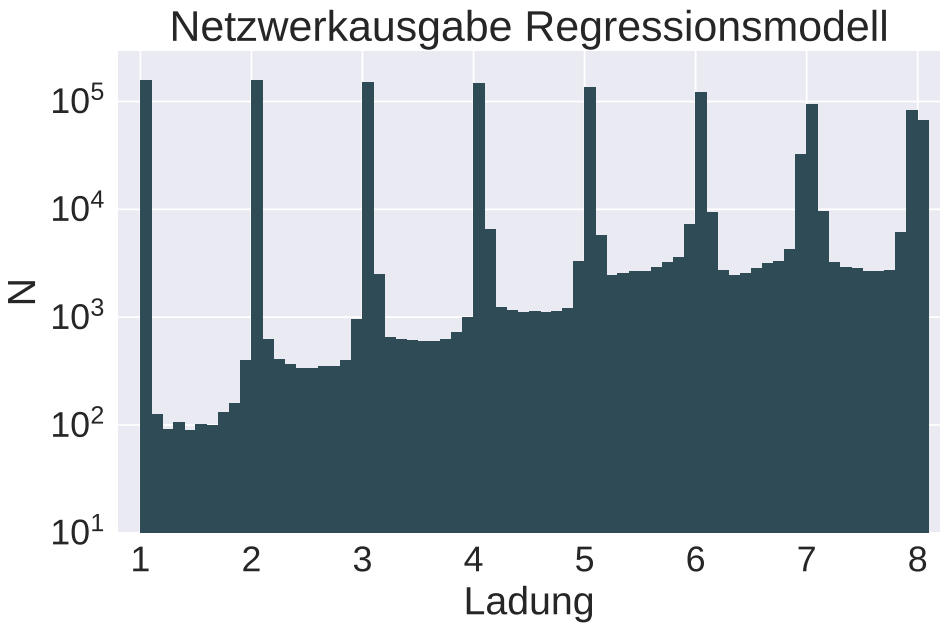

Abbildung 4.3: Netzwerkausgabeverteilung des besten Regressionsnetzwerks

Eine Methode Netzwerke zu verbessern liegt darin, bestimmte zusätzliche Abhängigkeiten, von denen die Ausgabe des Netzwerkes ebenfalls abhängen kann, zu begrenzen in dem man für jedes Bin der Abhängigkeit ein eigenes Netzwerk trainiert. Es wurden bessere Ergebnisse gesehen, wenn man das Netzwerk nach Rigiditäten zuschneidet[36] (Im folgenden R Teilung genannt). Außerdem rät ein Netzwerk, welches falsch rät, normalerweise nur leicht falsch (vergleich Abbildung 4.4), und ein Netzwerk, welches weniger als 8 Ladungen unterscheiden soll, tut sich auf diesen Ladungen besser als ein Netzwerk welches 8 Ladungen trennen muss. Folglich bietet sich eine weitere Aufteilung an (RQ Teilung): Je nachdem was ein erstes Netzwerk für eine Ladung approximiert, generiert ein anderes Netzwerk die beste Vorhersage[37].

[36] Auswahl der Rigiditätsbins in Anhang IV
[37] mehr dazu in Anhang IV

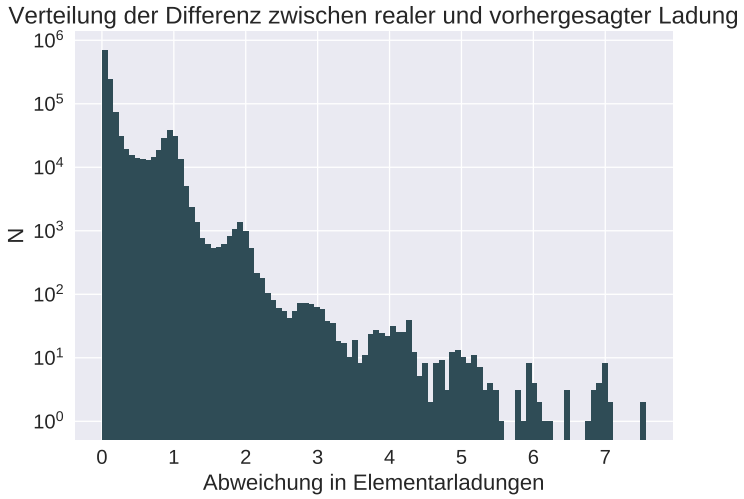

Abbildung 4.4: Histogramm der Abweichung einer vorhergesagten Ladung im Vergleich zur realen Ladung

Diese Methode hat den Nachteil anstatt einem Netzwerk, 46 oder 55 Netzwerke[38] zu benötigen, was deutlich mehr Zeit und Arbeitsspeicher benötigt um Werte vorherzusagen, und, zumindest in Python, nicht so schnell darin ist Werte vorherzusagen, da hier die Parallelisierung von Tensorflow wegfällt. Folglich werden im Folgenden ein nach Rigiditäten geteiltes Modell, ein nach Rigiditäten und Ladungen geteiltes Modell, und ein nicht geteiltes Modell gezeigt und der Anwendungsfall sollte entscheiden, was wichtiger ist: Genauigkeit oder Zeitkosten. Außerdem sollte man wohl erwähnen, dass der Preis dieser Methode ist, dass sich die Validierungsdaten mehr im Modell widerspiegeln können, also Testdaten wohl recht wichtig sind.

Quantisierung
Einer der interessanteren Effekte, die während des Trainings aufgetreten sind, ist wohl die Quantisierung der Ausgabe eines Regressions Modells (siehe Abb. 4.3). Ein Netzwerk, welches genügend Trainingsdaten hat (bei 100k Daten tritt nur teilweise Quantisierung auf, vergleich Abbildung 4.5), lernt selbstständig, dass Ladungen quantisierte Werte sind und gibt für jede Ladung praktisch nur noch einen Peak um den Mittelpunkt der Trainingsdaten dieser Ladung aus. Das kann so verstanden werden, dass in den Daten nicht genug Information steckt, um die Abweichungen von ganzen Ladungen vorherzusagen, und jede Variation in der Ausgabe theoretisch äquivalenter Eingabedaten nur Overfitting ist, was von der erhöhten Anzahl der Eingabedaten verhindert wird[39]. Dies ist ein nicht unbedingt erwünschter Effekt, aber auch ein nicht zu verhindernder Effekt, da die Erhöhung der Anzahl der Eingabedaten, nicht nur Δ durch Quantisierung senkt, sondern auch p verbessert. Sollten gaussverteilte Peaks benötigt werden, lassen sich diese z.B. aus der Mittelung des Klassifikationsnetzwerks generieren, wobei der Preis dafür Genauigkeit ist. Sollten nur Fehler benötigt werden, lassen sich diese aus den Wahrscheinlichkeiten des Klassifikationsnetzwerks generieren (vgl. Kapitel 4.4).

[38] vgl. Anhang IV und IV

[39] zum Test dieser Interpretation wurde ein Netzwerk dazu trainiert ganze Zahlen auf gaussverteilte Zufallszahlen um diese ganze Zahl abzubilden, hier tritt der selbe Effekt ein, vgl. Anhang IV

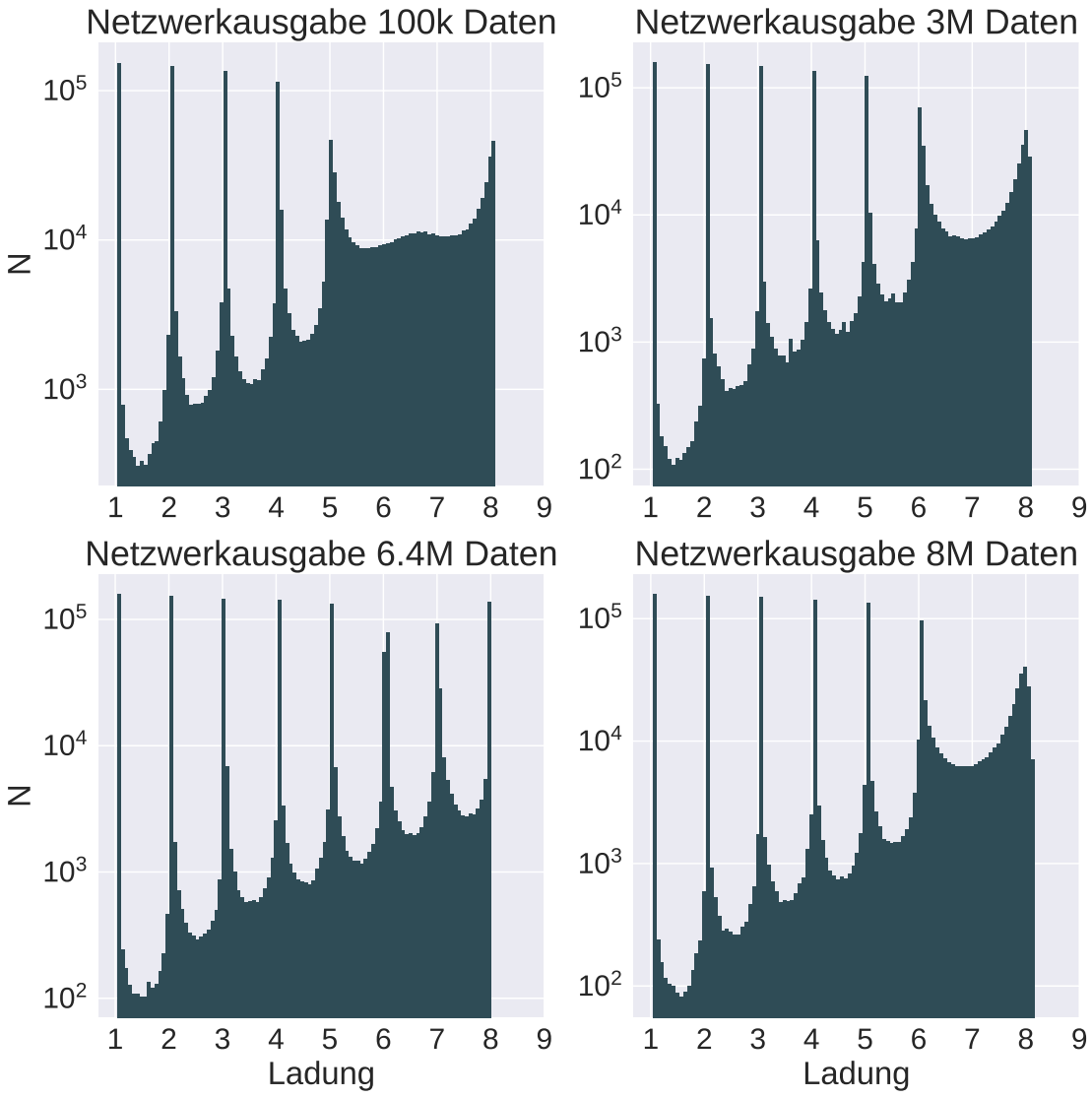

Abbildung 4.5: Ladungsausgabeverteilung für verschiedene Größen des Trainingssets, im Urzeigersinn von links oben: 100k, 3M, 6.4M, 8M. Die Tatsache das die Quantisierung für 8M wieder schlechter wird, liegt daran, dass eine solche Anzahl durch Wiederholung bestimmter Datenpunkte erreicht wird

Wahrscheinlichkeitsnormierung

Ein großes Problem bei der Vorhersage von Ladungen in der kosmischen Strahlung, ist die Tatsache, dass kleine Ladungen deutlich häufiger vorkommen, als große Ladungen. Das heißt wenn eine kleine Ladung eine kleine Chance hat, fehlerhaft als größere klassifiziert zu werden, aber diese deutlich häufiger vorkommt, ist die Chance, dass eine Identifikation als höhere Ladung wirklich die höhere Ladung ist, stark ab. Da hier aber immer nur die Wahrscheinlichkeiten dafür, dass eine Vorhersage der Wahrheit entspricht, optimiert werden (dies ist die Ausgabe eins Klassifikationsmodells und die Wahrscheinlichkeit, die in p eingerechnet wird), muss dies mit der Anzahl der Ladungen neu normiert werden. Schlussendlich werden deshalb auch jeweils zwei Graphen gezeigt, welche die Verwechslungswahrscheinlichkeiten zeigen. Man beachte, dass in den neu normierten Verwechslungsgraphen Ladungen, deren Vorkommen deutlich kleiner ist

als die der vorherigen Ladung, auswaschen. Hier tritt dies besonders bei $q = 3$ und $q = 7$ auf.

Bewertung

Die Qualität der Regressionsmodelle gemittelt ist in Tabelle 4.1 zu sehen, aufgeteilt nach Soll-Ladung und Rigidität in Abb. 4.7 und Abb. 4.8. Außerdem ist eine Verwechslungsmatrix der einzelnen Ladungen, normiert und nicht normiert, in 4.6 zu sehen.

Index	keine Teilung	R Teilung	R und Q Teilung	Bethe Bloch Modell
Δ	0,2224	0,2070	0,2049	0,3966
p	16,88%	15,05%	15,05%	25,47%
χ	135	118	123	228
Vorhersagezeit C++	143μs	144μs	296μs	160ns
Vorhersagezeit Python	79μs	2,1ms	5,8ms	934μs

Tabelle 4.1: Bewertungsindices Regressionsmodell gemittelt über alle Ladungen und Rigiditäten

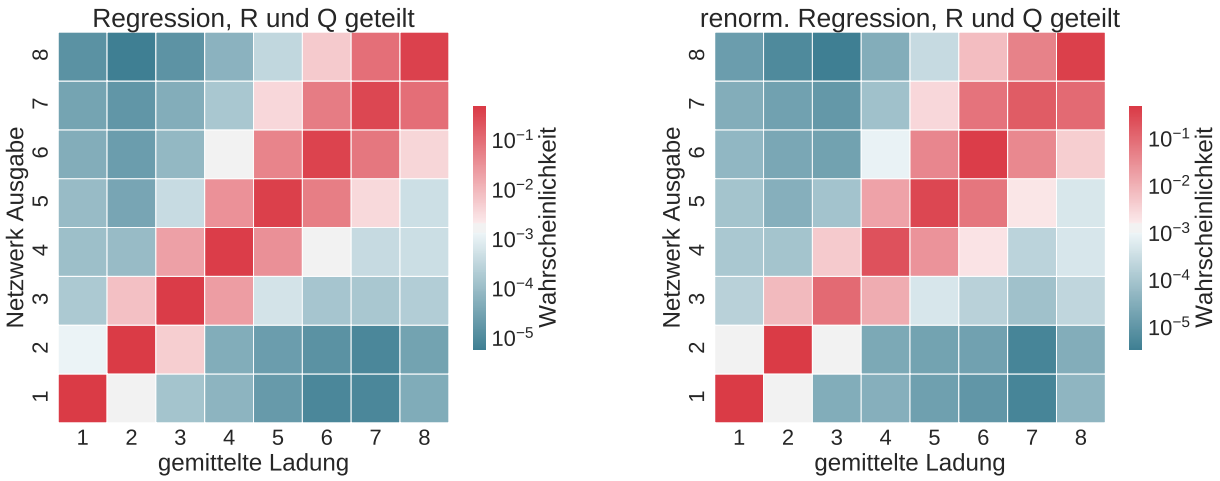

Abbildung 4.6: reale Ladung gegen ausgegebene Ladung des besten Regressionsmodells, links nach gleichverteilten Ladungen, rechts mit Wahrscheinlichkeitsrenormierung

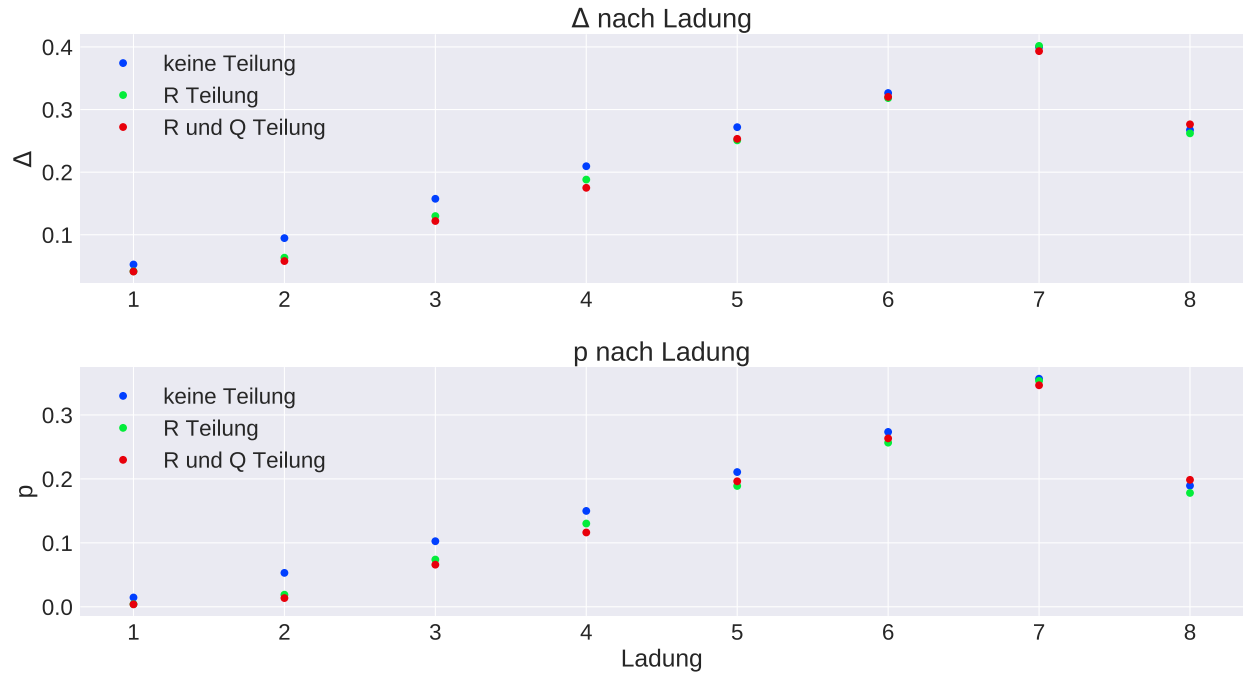

Abbildung 4.7: Indizes eines Regressionsmodells nach Eingabeladung aufgeteilt

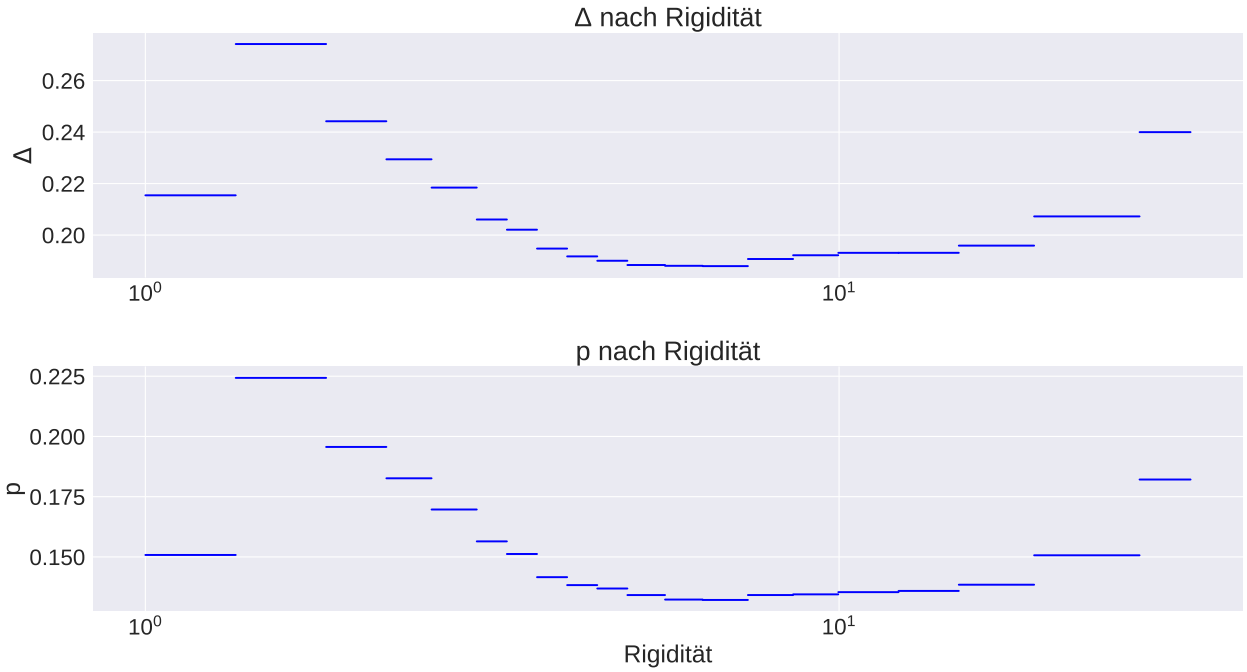

Abbildung 4.8: Indizes des besten Regressionsmodells nach Rigiditätbin aufgeteilt

4.4 Das Klassifikationsmodell

Da die Ausgabe eines Regressionsmodells sowieso fast quantisiert ist, hat ein Klassifikationsmodell praktisch keine Nachteile mehr. Der Vorteil hier ist, dass ein Klassifikations Modell nicht nur die wahrscheinlichste Ladung, sondern auch die Wahrscheinlichkeit für jede Ladung

ausgibt[40](eine Beispielausgabe eines Klassifikationsmodell ist in Abb. 4.9 zu sehen) [41]. Das Klassifikations Modell benutzt dieselbe Datenoptimierung, hat aber eine eigene Hyperparameteroptimierung[42]. Auch hier gibt es wieder geteilte Modelle und ein nicht geteiltes schnelleres Modell. Zum Vergleich mit dem Regressionsmodell werden Indizes, welche einen genauen Ladungswert benötigen (Δ,χ), mit einem gewichteten Mittelwert als Ladung berechnet, wobei die Gewichte den Wahrscheinlichkeiten entsprechen, während p mit der höchsten Wahrscheinlichkeit berechnet wird[43]. Die Ausgabeverteilung des besten Klassifikations Netzwerk ist in Abb. 4.10 zu sehen.

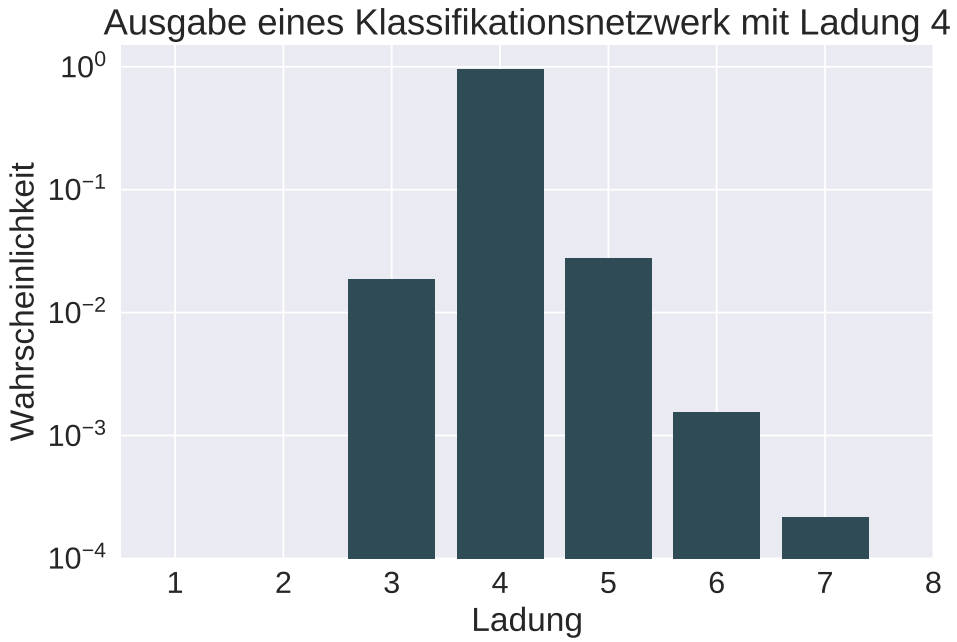

Abbildung 4.9: Beispielausgabe eines Regressionsnetzwerk

[40]Ein Beweis, dafür dass die Wahrscheinlichkeit, für den Fall indem ein ausgegebener Wert mit einer angegebenen Wahrscheinlichkeit p falsch sei, $1-p$ sei, ist in Anhang V Bilder V.1,V.2,V.3 zu finden.

[41]Dies ist außerdem besonders interessant, da die Fehlerwahrscheinlichkeiten eine gute Approximation für den Fehler des Ausgegebenen Wertes liefern, da die Abweichungen von der realen Ladung in dem meisten Fällen eins ist und somit unter Vernachlässigung der benutzen Freiheitsgerade $\frac{X^2}{ddof} = \frac{1}{N} \cdot (N_{richtig} \cdot (\frac{0}{\sigma})^2 + N_{falsch} \cdot (\frac{1}{\sigma})^2)$ von $\sigma = \sqrt{\frac{N_{falsch}}{N}}$ zu eins gelöst wird. Folglich entspricht die Varianz der Verteilung eines in etwa der Fehlerwahrscheinlichkeit.

[42]vgl. Anhang II und III

[43]es hat sich gezeigt, dass diese Vorgehensweise etwas bessere Werte für p erreicht, als hier auch die Ladung zu mitteln

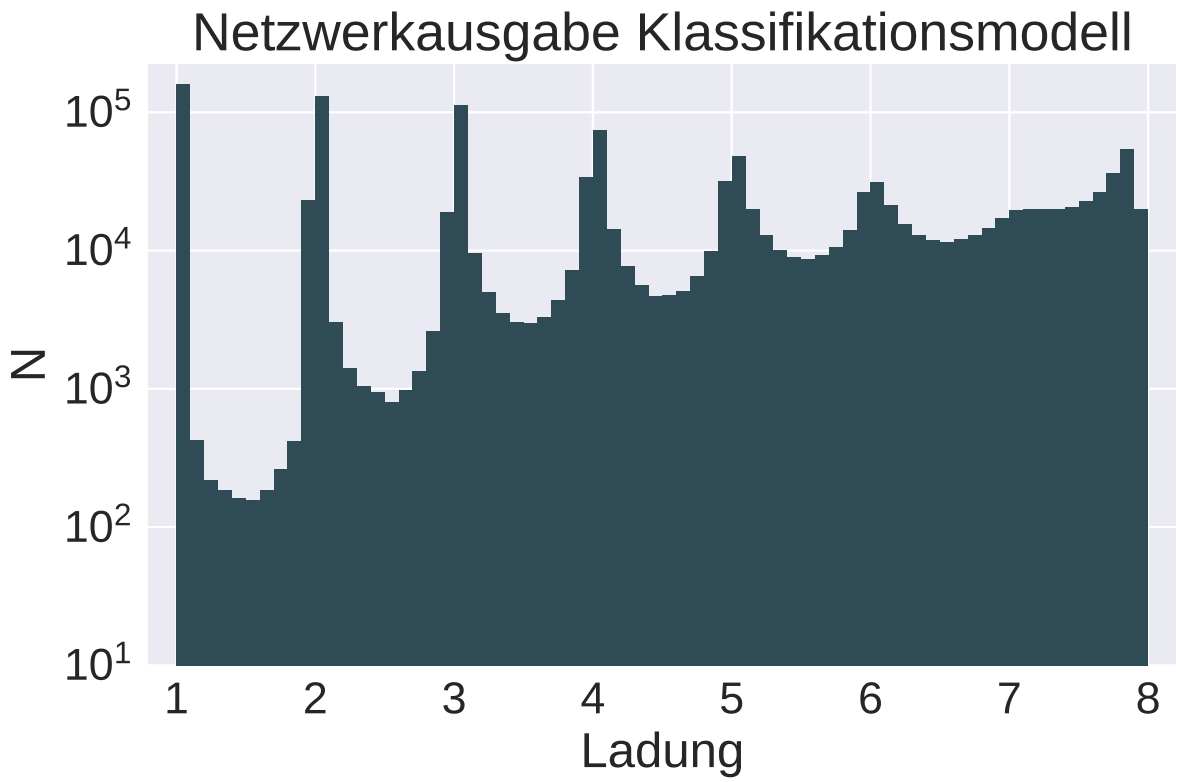

Abbildung 4.10: Netzwerkausgabeverteilung der gemittelten Ausgabe des besten Klassifikationsnetzwerk, man beachte die schwächere Quantisierung für große Ladungen, aber insbesondere das hier die Quantisierung der kleinen Ladungen wirklich bedeutet, dass diese richtig vorhergesagt wurden

Bewertung

Die Qualität der Regressionsmodelle gemittelt ist in Tabelle 4.2 zu sehen, aufgeteilt nach Soll-Ladung in Abb. 4.12 (die Teilung nach Rigiditäten sieht aus wie beim Regressionsnetzwerk). Außerdem ist wieder eine Verwechslungsmatrix der einzelnen Ladungen, dieses Mal mit einem höheren Binning und mit begrenzter Nachselektion (siehe Ladungsverteilung in Bild IV.3), in 4.11 zu sehen. Beispielhaft die Ausgabe des Klassifikationsmodells von Events mit Ladung 3, sowie die Verteilung der Eingabeladungen, welche eine Ausgabe von 3 produzieren, einmal ohne Schnitt und einmal mit einem Schnitt auf dessen Wahrscheinlichkeit, sind in Abb. 4.13, 4.14 und 4.15 zu sehen.

Index	keine Teilung	R Teilung	R und Q Teilung	Bethe Bloch Modell
Δ	0,2672	0,2378	0,2445	0,3966
p	19,37%	15,37%	16,17%	25,47%
χ	134	114	118	228
Vorhersagezeit C++	155μs	150μs	555μs	160ns
Vorhersagezeit Python	74μs	1,9ms	4,9ms	934μs

Tabelle 4.2: Bewertungsindices Klassifikationsmodell gemittelt über alle Ladungen und Rigiditäten

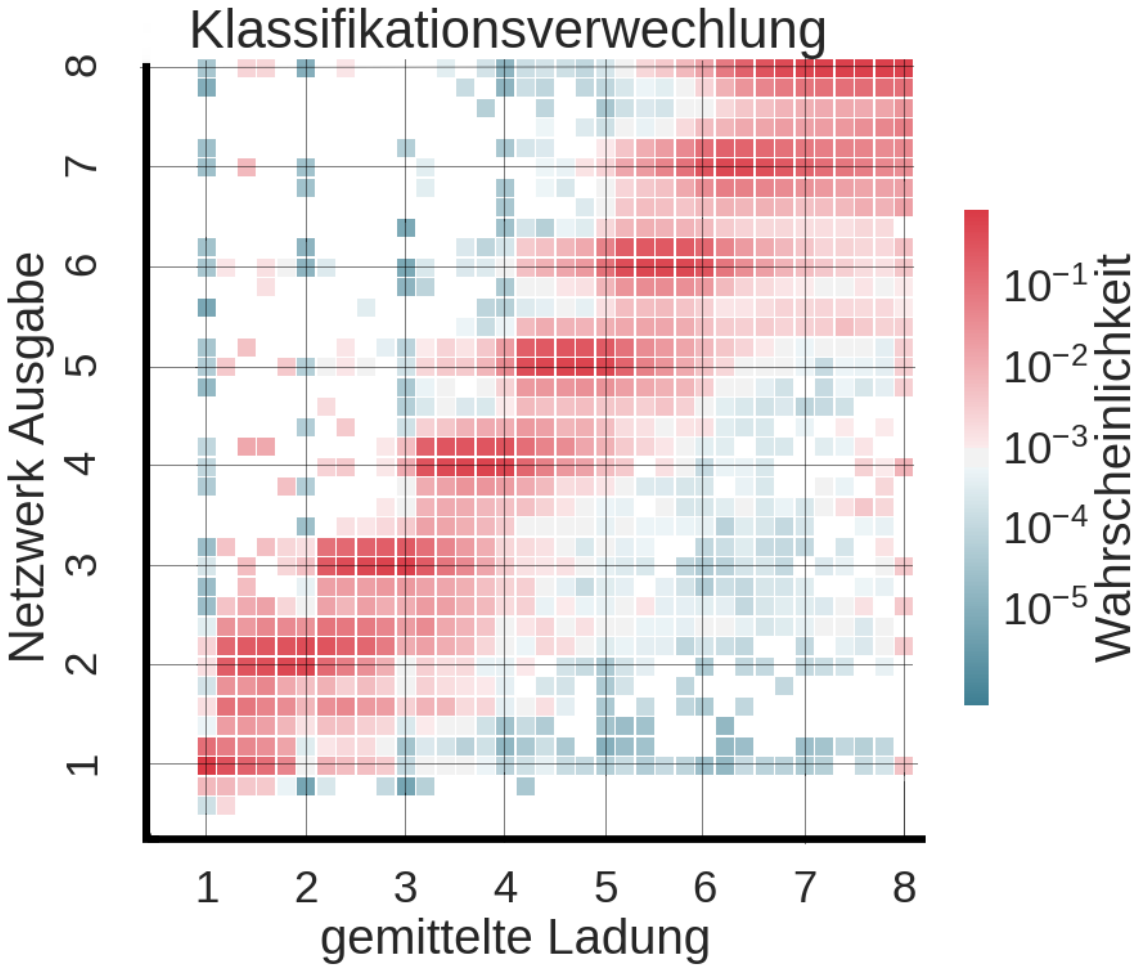

Abbildung 4.11: reale Ladung gegen ausgegebene Ladung des besten Klassifikationsmodells, hierbei ist die schlechte Statistik nur ein Produkt fehlerhafter Klusterfunktionsweise und begrenzter Zeit

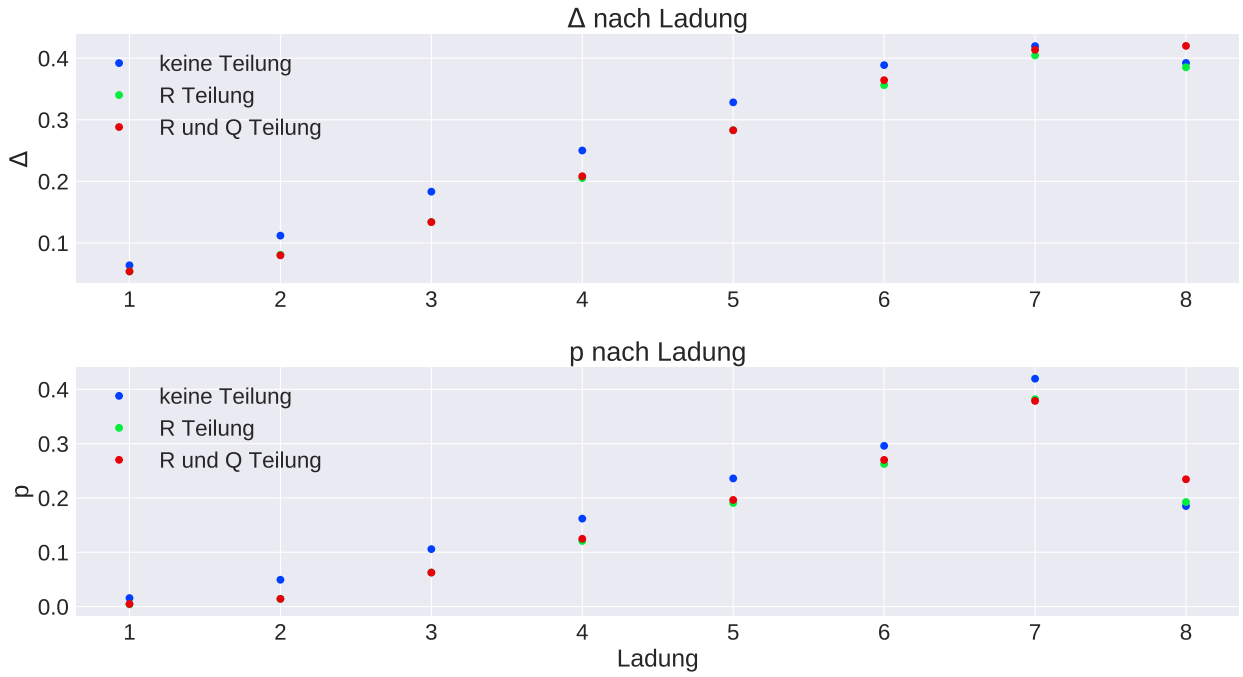

Abbildung 4.12: Indize eines Klassifikationsmodells nach Eingabeladung aufgeteilt

Um zu begründen, wieso Klassifizierungsnetzwerke schlechtere Ergebnisse liefern als Regressionsnetzwerke, wurde ein Regressionsnetzwerk trainiert, um nicht die Ladung, sondern den gerundeten Wert der Ladung vorherzusagen. Dieses liefert ähnlich schlechtere Werte wie das dazugehörige Klassifikationsnetzwerk. Dies legt nahe, dass nicht gerundete Ladungen eine Art Gewichtung liefern, welche mögliche Fehlzuordnungen in den Trainingsdaten weniger stark wertet und somit durch das Verhindern von Overfitting die Qualität des Netzwerks erhöht. Dies validiert vor allem den letzten Cut in der Nachselektion(Kapitel 3.1.3)

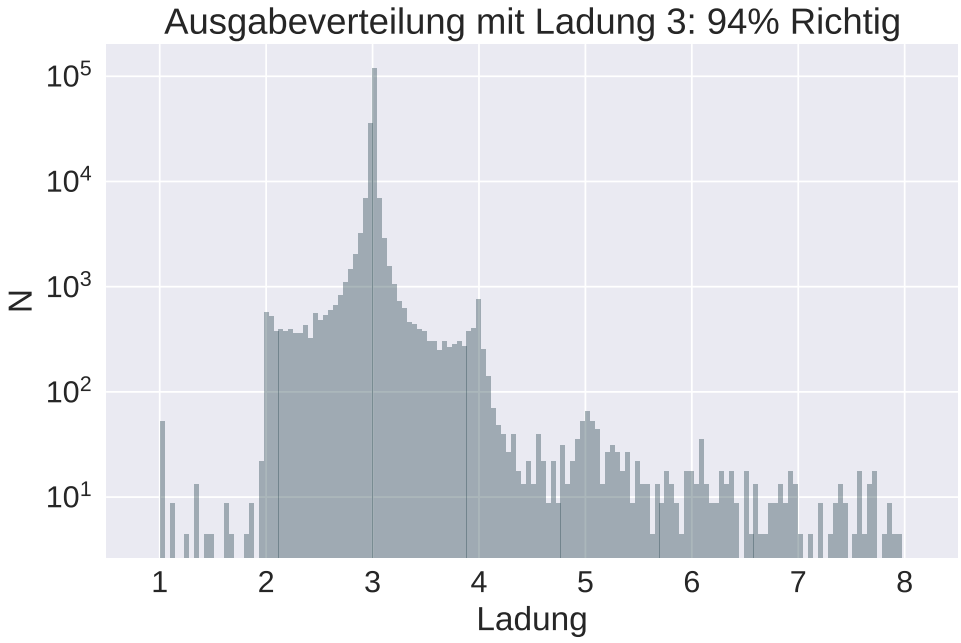

Abbildung 4.13: Ausgabeverteilung des Klassifikationsmodells für Events welche eine Ladung von 3 besitzen

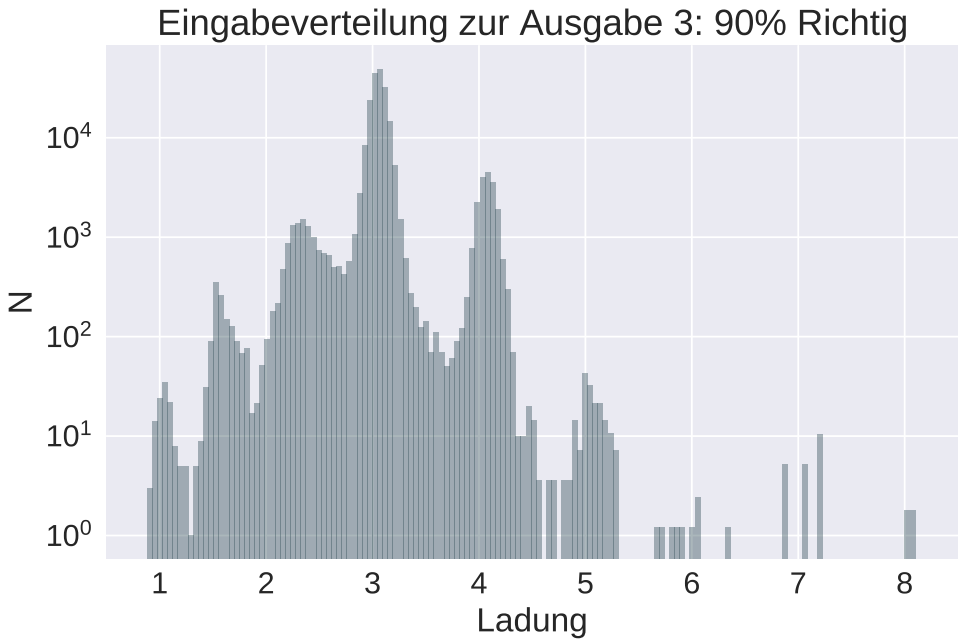

Abbildung 4.14: Eingabeladungsverteilung der Events welche eine Ausgabe von 3 generieren

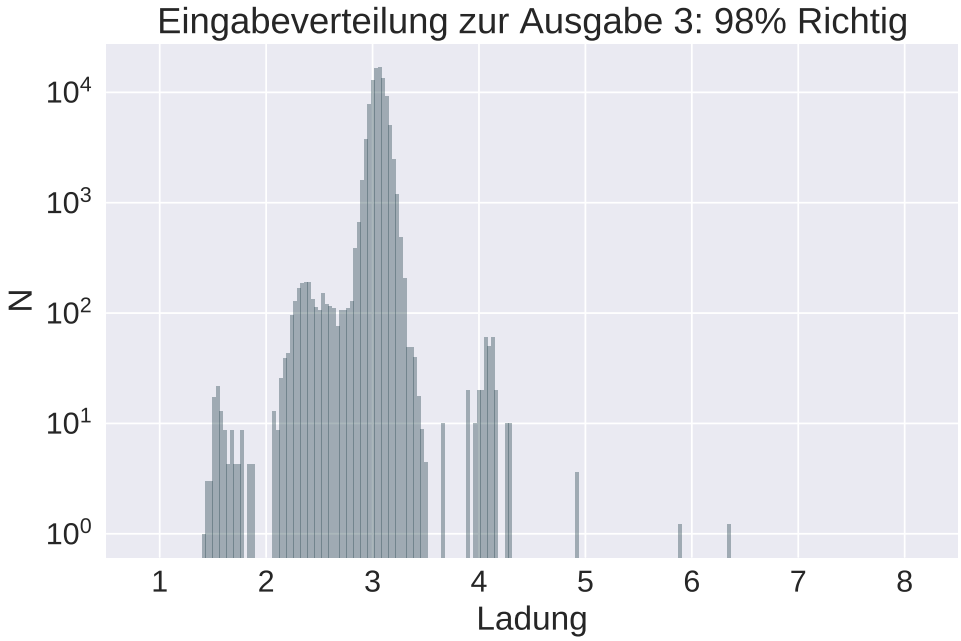

Abbildung 4.15: ebenfalls eine Eingabeladungsverteilung der Events, welche eine Ausgabe von 3 generieren, nur das hier nur solche Events gewählt werden, welche nach Netzwerkausgabe eine Wahrscheinlichkeit von 97% oder höher haben als 3 Klassifiziert zu werden

Abb. 4.15 zeigt das das Klassifikationsmodell noch weit bessere Ergebnisse erlaubt in dem man zu Kosten der Datenzahl auf die Wahrscheinlichkeit schneidet, um die Genauigkeit zu erhöhen.

4.5 Vergleich der Modelle

Ein Vergleich der nicht geteilten und am besten geteilten Modelle ist in Tabelle 4.3 und Tabelle 4.4, sowie Abb. 4.16 zusehen.

Index	ungeteilte Regression	ungeteilte Klassifikation
Δ	0,2224	0,2672
p	16,88%	19,37%
Vorhersagezeit C++	143μs	155μs
Vorhersagezeit Python	79μs	74μs

Tabelle 4.3: Vergleich der ungeteilten Modelle

Index	beste Regression	beste Klassifikation
Δ	0,2049	0,2378
p	15,05%	15,37%
Vorhersagezeit C++	296μs	150μs
Vorhersagezeit Python	5,8ms	1,9ms

Tabelle 4.4: Vergleich der besten Modelle

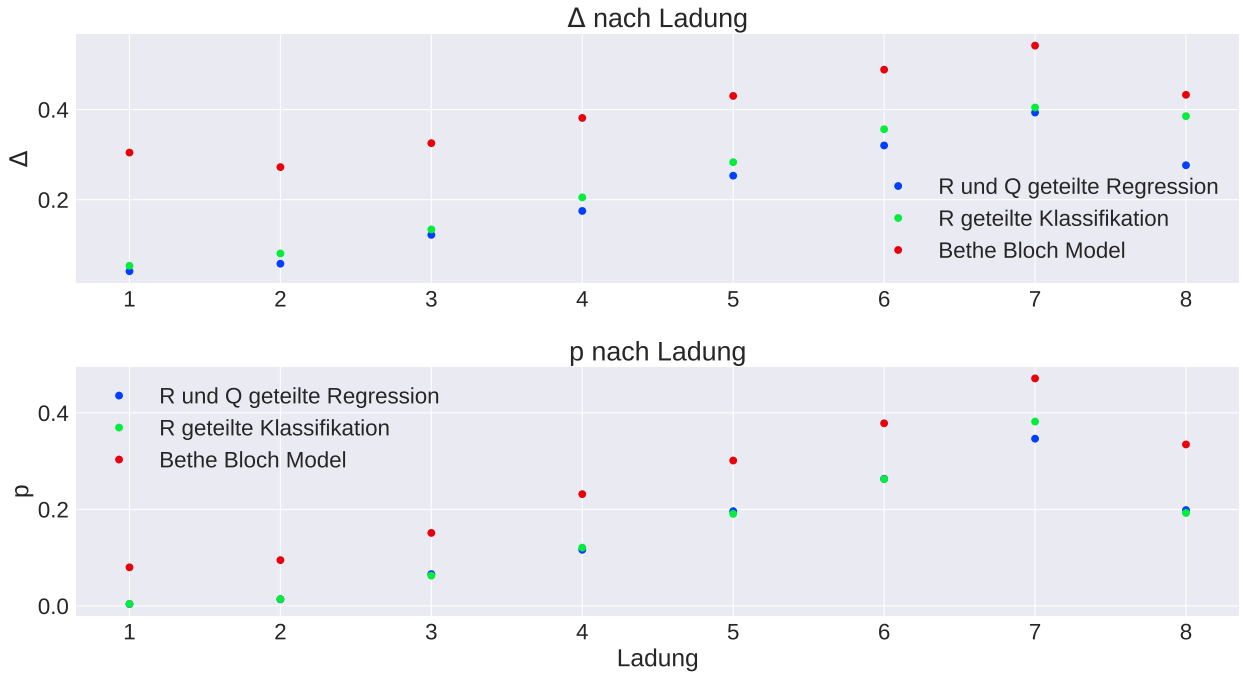

Abbildung 4.16: Modellindizes nach Ladung aufgeteilt

Auch wenn ein Klassifikationsmodell im ganzen etwas schlechtere Indizes besitzt, bietet es doch Vorteile, folglich ist die Frage, welches Modell zu benutzen sei eine Frage des Anwendungsfalls.

Testdaten

Tests der Modelle auf Testdaten ergeben zwar andere Ergebnisse als der Test dieser auf Validierungsdaten (Ein Beispiel der Indices auf beiden Datensets ist in Abb. 4.17 zu sehen), aber der Unterschied beträgt konstant weniger als 0,001 und die Richtung variiert nach Modell. Da außerdem, bei einer Testsetgröße von circa 0,32M Elementen, die Poisson Verteilung einen Fehler von circa $\frac{1}{\sqrt{320000}} = 0,0018$ für p generiert, und der Unterschied ohne Neuronale Netzwerke sogar größer ist, ist dies wohl akzeptabel.

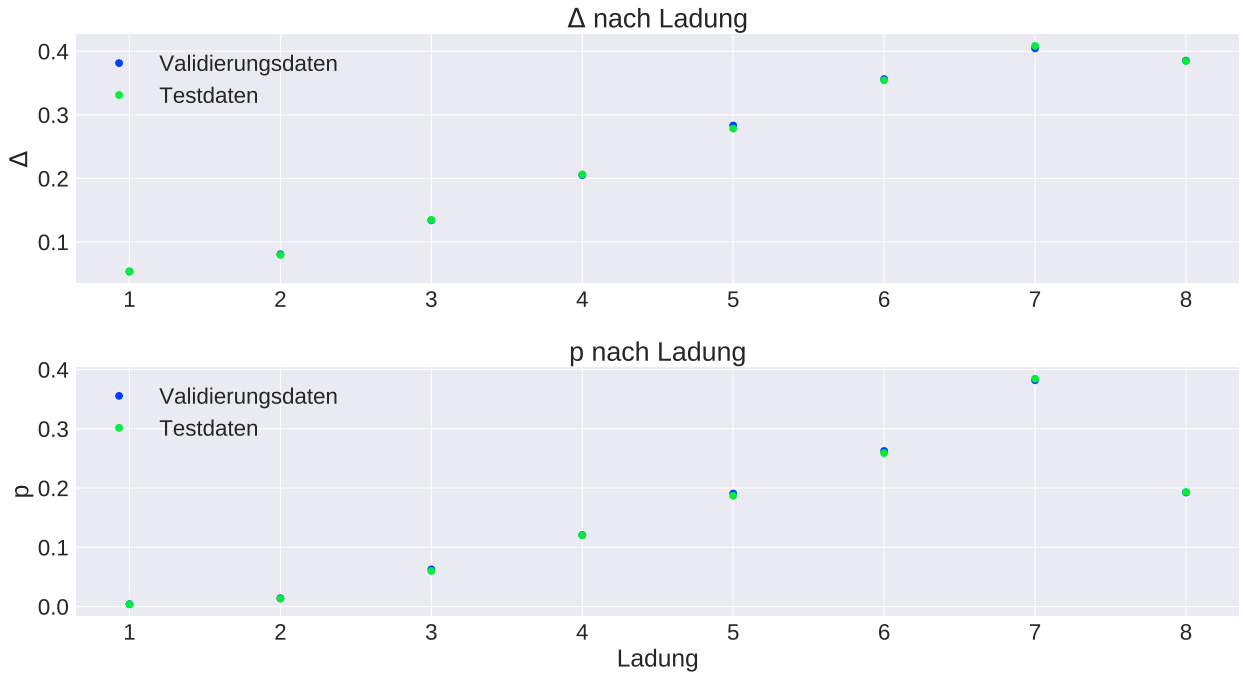

Abbildung 4.17: Vergleich von Test und Validierungsdaten nach Ladung des besten Klassifikationsnetzwerks, weitere Vergleiche siehe Anhang VI.1

5 Fazit

Es wurde gezeigt, dass es möglich ist, ein Neuronales Netzwerk zu trainieren, um die Ladung in einem Übergangsstrahlungsdetektor zu bestimmen, und dass diese Ladungsbestimmung deutlich besser ist, als die Alternative ohne Neuronales Netzwerk: Während das Bethe Bloch Modell eine Chance von mehr als 25% hat ein Event falsch zu klassifizieren, hat die beste Version mit Neuronalen Netzwerken nur eine Fehlerchance von nur circa 15%. Insbesondere nützlich ist dies, wenn man die unterschiedlichen Ladungszahlen, aus denen kosmische Strahlung besteht, rausrechnet. Hier klassifiziert das Netzwerk nur 1 von 176 Events falsch, während ohne Neuronale Netzwerke 1 von 12 Events falsch klassifiziert wird[44]. Schlussendlich zeigt Abb. 5.1 dass, das die Ausgabe des Netzwerks auf bei deutlich weniger zugeschnittenen Daten sicher Ladungspeaks unterscheiden kann, während die AMS Peaks insbesondere bei 3 und 4 fast nicht mehr vom Hintergrund zu unterscheiden sind. Insbesondere gilt dies, da die ausgegebene Wahrscheinlichkeit eines Klassifikationsnetzwerks es ermöglicht, sich nur Events anzusehen, bei denen die Ladungszuordnung praktisch beliebig sicher ist.

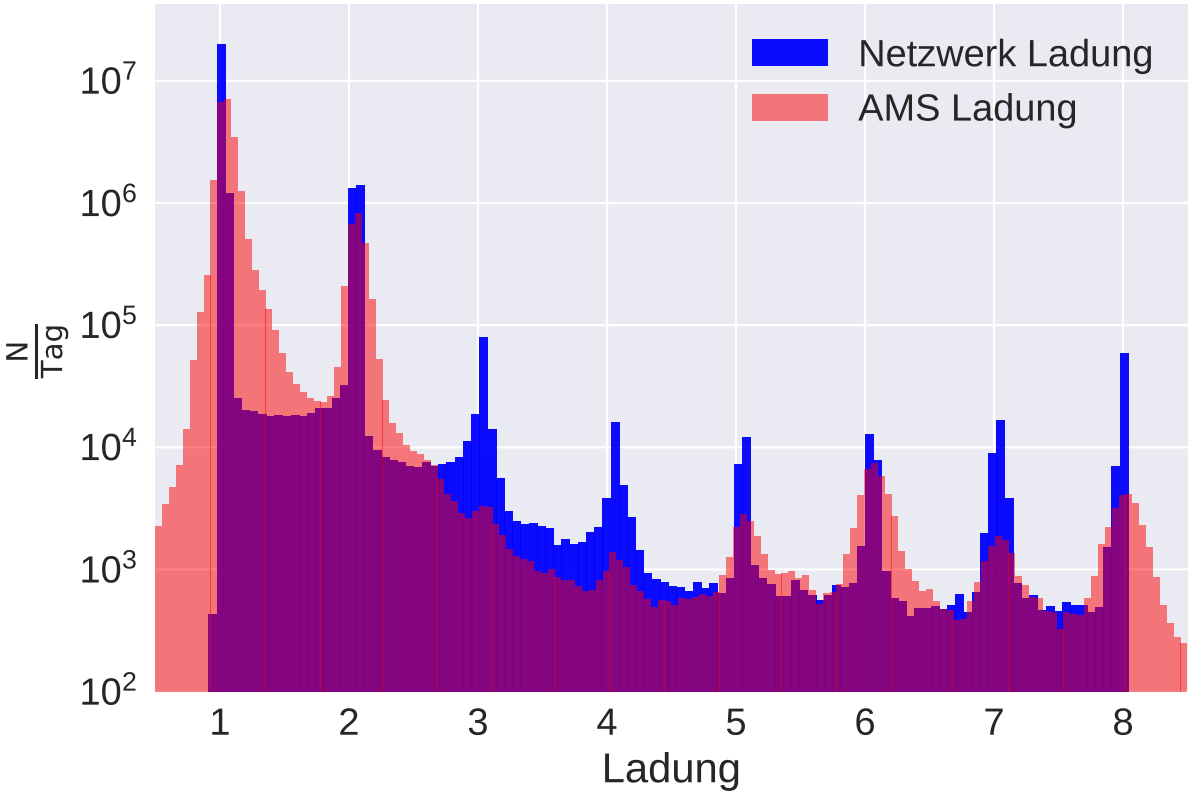

Abbildung 5.1: Ladungsverteilung des Netzwerks gegen die Ladungsverteilung welche aus Tracker und ToF gemittelt wurde, ohne Nachselektion auf den Daten eines Tages

[44]Dies ist nur leicht besser, als ein Modell welches immer nur 1 rät, dieses würde 1 von 9 Events falsch klassifizieren

5.1 Ausblick

Hier wurden nur grundlegende Optimierungen getroffen, aber es sollte durchaus möglich sein, die Geschwindigkeit des Netzwerks zu Kosten von ein paar Promille Genauigkeit zu erhöhen, da die meisten der probierten Optimierungen nur darauf abzielen, höhere Genauigkeiten zu erreichen. Die größte Möglichkeit, Unterscheidungsgenauigkeiten zu verbessern, ist es aber, die Anzahl der zu trainierenden Ladungen zu senken (dies verbessert auch die Genauigkeit der bestehenden Ladungen). Dies erlaubt auch, das Problem, welches durch die Tatsache erzeugt wird, dass Wahrscheinlichkeiten (und die damit verbundenen Genauigkeiten) sehr stark von der variierenden Zahl der Eingabeladungen verzerrt werden, zu minimieren. Alternativ lassen sich auch Netzwerke für höhere Ladungen trainieren, dann mit gewichteten Eingangsdaten, auch wenn dies die Netzwerk-Genauigkeit deutlich senken wird. Insbesondere könnten Klassifikationsnetzwerke auch nicht zusammenhängende Ladungen oder gar Gruppen von Ladungen unterscheiden. Schlussendlich ist es natürlich auch noch durchaus möglich die Genauigkeit auf den hier benutzten Daten zu optimieren. Eine Auflistung einiger Möglichkeiten dazu, folgt in Tabelle 5.1.Hierbei sollte man immer bedenken, dass sämtliche Zahlen nur sehr grobe Schätzungen sind, sowie dass der Preis einer jeden Erweiterung Data Leakage ist (da ein Modell nur dann benutzt wird, wenn es bessere Werte liefert).

Vorschlag	Aufwand	Nutzen	Data leakage
Zufällige Initialisierung, nehme das bestes Modell	Beliebig	$O(10^{-4} \cdot \sqrt{\frac{t}{1\,\text{Tag}}})$	Sehr Hoch
Hyperparameteroptimierung für jedes der Teilmodelle	2 Tage	$O(10^{-3})$	Hoch
Hyperparameteroptimierung auf großem Datenset	2 Wochen	$O(10^{-4})$	Mittel
Die Stärke der Precuts optimieren	4 Wochen	max $O(10^{-3})$	Gering
Training auf Monte Carlo Daten(da mehr Daten)	1-2 Monate	$O(10^{-2})$	Mittel
Konvolutions anstatt Tracker Spur	1-3 Monate	$O(10^{-3})$	Indirekt
Iterative Hyperparameteroptimierung	2 Tage	$O(10^{-4})$	Mittel
Evolutionäre Optimierung	Beliebig	$O(10^{-4} \cdot \sqrt{\frac{t-t_0}{1\,\text{Woche}}})$	Riesig
Rigiditätsbins komplett testen	1 Woche	$O(10^{-3})$	Mittel
Ladungsgruppen für Q-Splits vorhersagen	3 Tage	$O(10^{-4})$	kaum
Anzahl und Position der Dropout Layer optimieren	1 Woche	$O(10^{-4})$	Mittel

Tabelle 5.1: Vorschläge zu möglichen Verbesserungen

Simulationen wurden durchgeführt mit Ressourcen, welche von der Universität RWTH Aachen unter Projektbezeichnung thes0450 gewährt wurden

Anhang

I Implementation

Als Implementation in C++ dienen zwei Klassen, welche momentan *regressionpredicter* und *classificationpredicter* heißen. Diese werden mit einem Pfad zum Ordner in dem die jeweiligen Modelle liegen initialisiert und stellen jeweils Modelladefunktionen und Vorhersagefunktionen zur Verfügung, um ein nicht geteiltes, ein nach Rigidität geteiltes und ein nach Rigidität und Ladung geteiltes Netzwerk zu laden (load_nosplit(), load_rsplit(), load_rqsplit()) und solche um ein Array von Eingangsdaten mit Länge 110 und je nach Methode noch einen Rigiditätswert, auf eine Ladung(Regressionssystem) oder auf ein Array von Ladungswahrscheinlichkeiten mit Länge 8(Klassifikationssystem) abzubilden (predict_nosplit(double* x), predict_rsplit(double* x, double r), predict_rqsplit(double* x, double r)). Außerdem gibt es eine Funktion predict(double* x, double r), welche automatisch die geladene Vorhersagefunktion auswählt, und, sollte noch kein load ausgerufen worden sein, automatisch die nach Rigiditäten geteilten Modelle lädt. Selbiges tun auch andere Vorhersagefunktionen, wobei sie natürlich die für ihre Ausführung benötigten Modelle laden. Dies funktioniert nicht, wenn schon ein anderes Modell geladen ist, aber eine Klasse, welche kompliziertere Vorhersagen treffen soll, kann auch einfachere Vorhersagen treffen. Als Eingabe benötigen, neben der Rigidität für geteilte Modelle, alle Vorhersagefunktionen ein Array von 110 Gleitkommazahlen, welche von double** getlis(Event event) generiert werden. Die Ausgabe von getlis ist ein Array von Arrays, hierbei ist das erste Array die gesuchte Eingabe, während das zweite Array nur die Rigidität enthält. Sollte die Rigidität nicht bekannt sein, fallen alle Vorhersagefunktionen für negative Eingabewerte der Rigidität auf nicht geteilte Netzwerke zurück. Außerdem existiert eine Implementation des nicht neuronalen Netzwerkes, namens betheblochpredicter, welche ebenfalls dasselbe Eingabearrayformat benutzt. Ich empfehle eine Vorhersageklasse am Anfang des Programms zu erstellen, und somit das zu benutzende Netzwerk nur hier zu laden, da das einlesen der Netzwerke ein wenig dauern kann.

Teilung	Ladezeit
keine teilung	15ms
R geteilt	100ms
R und Q geteilt	1s

Tabelle I.1: Einlesedauer verschiedener Vorhersagetypen

Der Grund für die Tatsache, dass Python teilweise schnellere Vorhersagen ermöglicht, liegt daran, dass die Vorhersage in Tensorflow von Natur aus parallelisiert ist (also mehrere Events gleichzeitig verarbeitet). Dies ließe sich natürlich auch in C++ implementieren, bringt aber wenig, wenn Events einzeln verarbeitet werden.

Typ	Teilung	Vorhersagezeit C++ in μs
Regression	keine	143
Regression	R	144
Regression	RQ	296
Klassifikation	keine	155
Klassifikation	R	150
Klassifikation	RQ	555
Bethe Bloch	keine	0,16

Tabelle I.2: Vorhersagezeiten nach Modelltyp und Teilung

Außerdem sollte beachtet werden, dass die Ausgabe der Klassifikationsmodelle nicht nach Ladungszahlen normiert ist (vgl. Kapitel 4.3), da die Ladungszahlen je nach Cut durchaus variieren können.

II Hyperparameteroptimierung

II.1 Einfache Optimierung

Die einfachste Version, Hyperparameter zu optimieren ist es wohl, diese einfach auszuprobieren. Da aber aus Zeitgründen, in jedem Optimierungsschritt nur ein Parameter verändert wird, ist diese Methode recht stark vom anfänglichen Modell abhängig. Außerdem liegt dieser Methode eine gewisse Monotonie zu Grunde, welche ebenfalls durch starke Hyperparameterkorrelationen zerstört werden kann, also ein Minima was bei einer bestimmten Stapelgrößen oder eine bestimmten Dimension nicht minimal ist, aber bei dieser Dimension und Stapelgröße sein globales Minima hat, wird dieses niemals erreichen[45]. Trotzdem scheint diese Methode recht sinnvolle Hyperparameter zu finden. Alternativ wäre es auch möglich, die Methode iterativ anzuwenden (man fange mit einem Modell an, findet für dieses optimale Hyperparameter und tut dann dasselbe für die neuen Hyperparameter), das wird hier allerdings nicht angewendet. Hier die Ausgabe der Hyperparameteroptimierung beispielhaft für das Regressionsnetzwerk

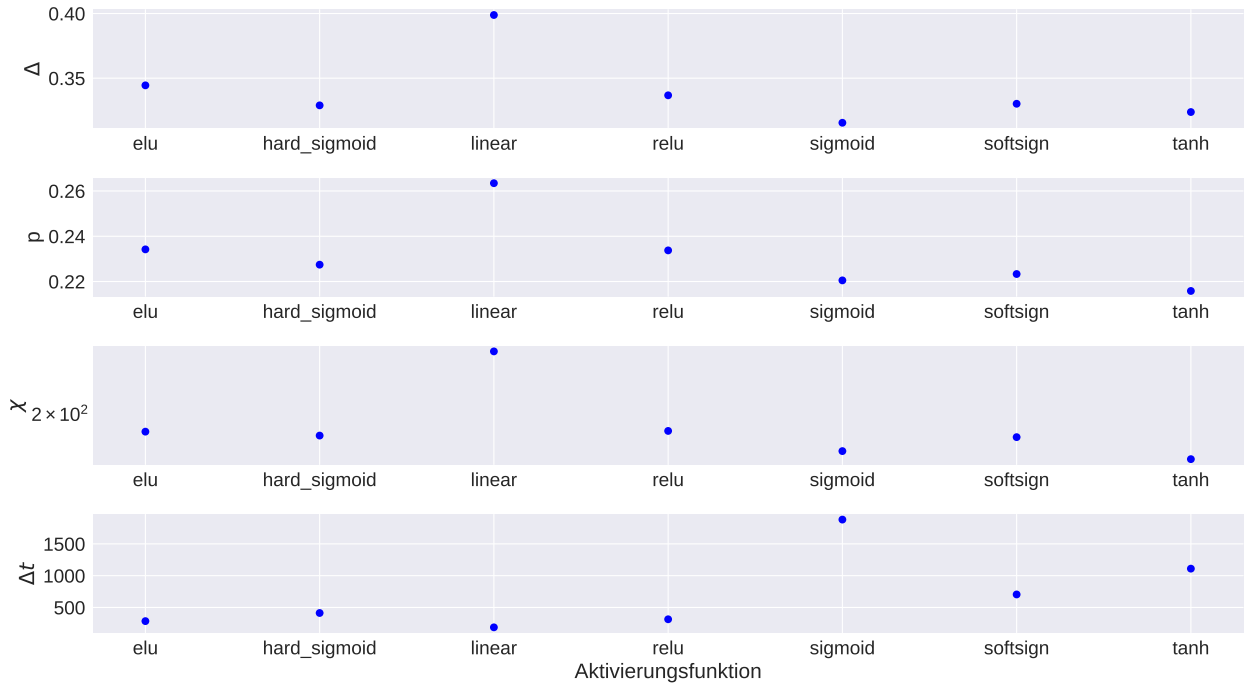

Abbildung II.1: Hyperparameteroptimierung nach Aktivierungsfunktion

[45] Das kann durchaus ein positiver Effekt sein, da er unter Umständen Data Leakage senkt.

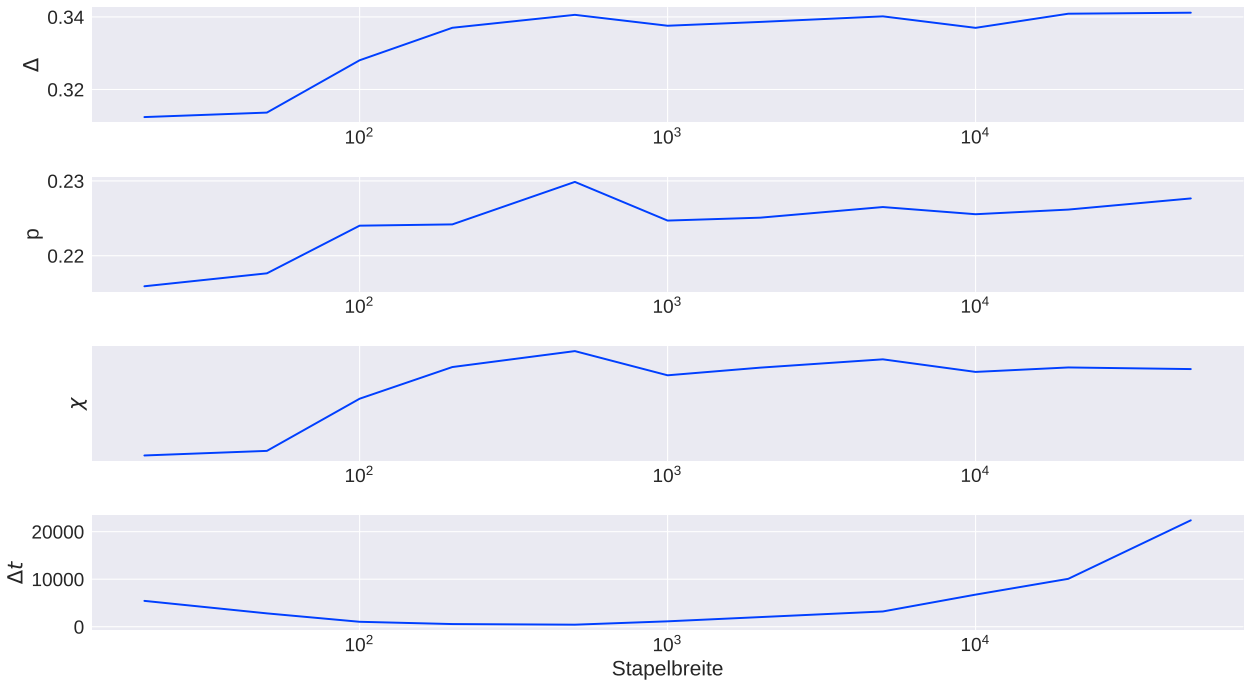

Abbildung II.2: Hyperparameteroptimierung nach Stapelgröße

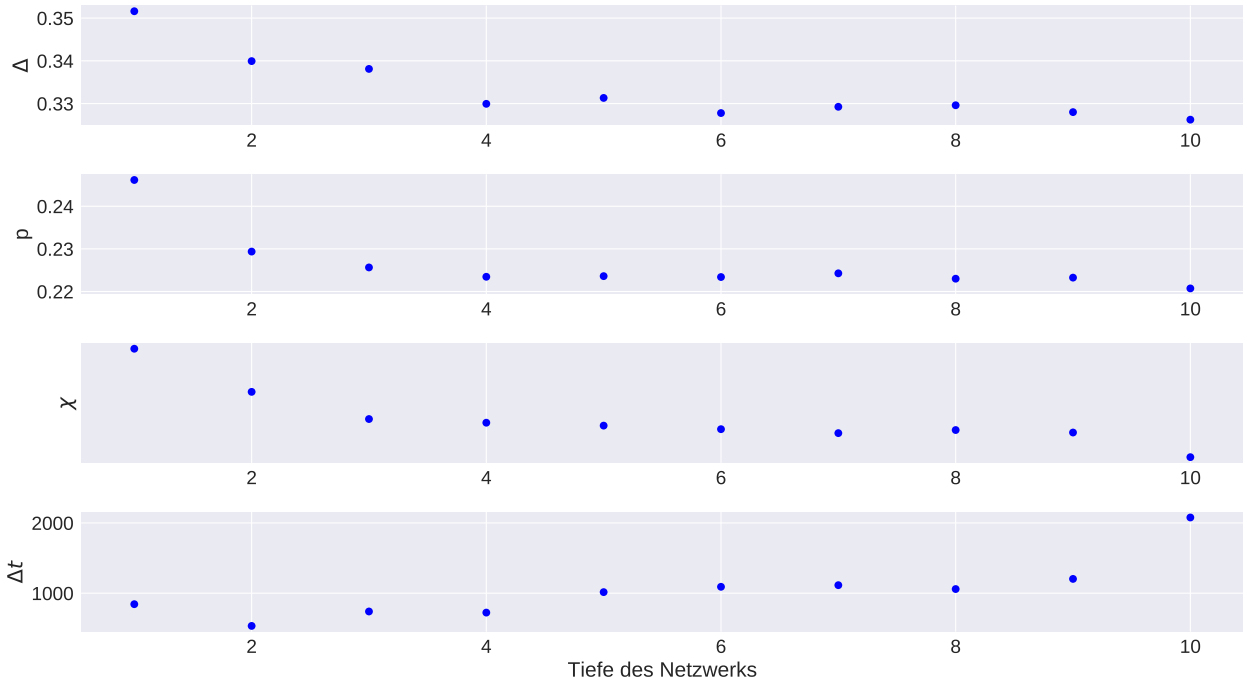

Abbildung II.3: Hyperparameteroptimierung nach Anzahl der Layer

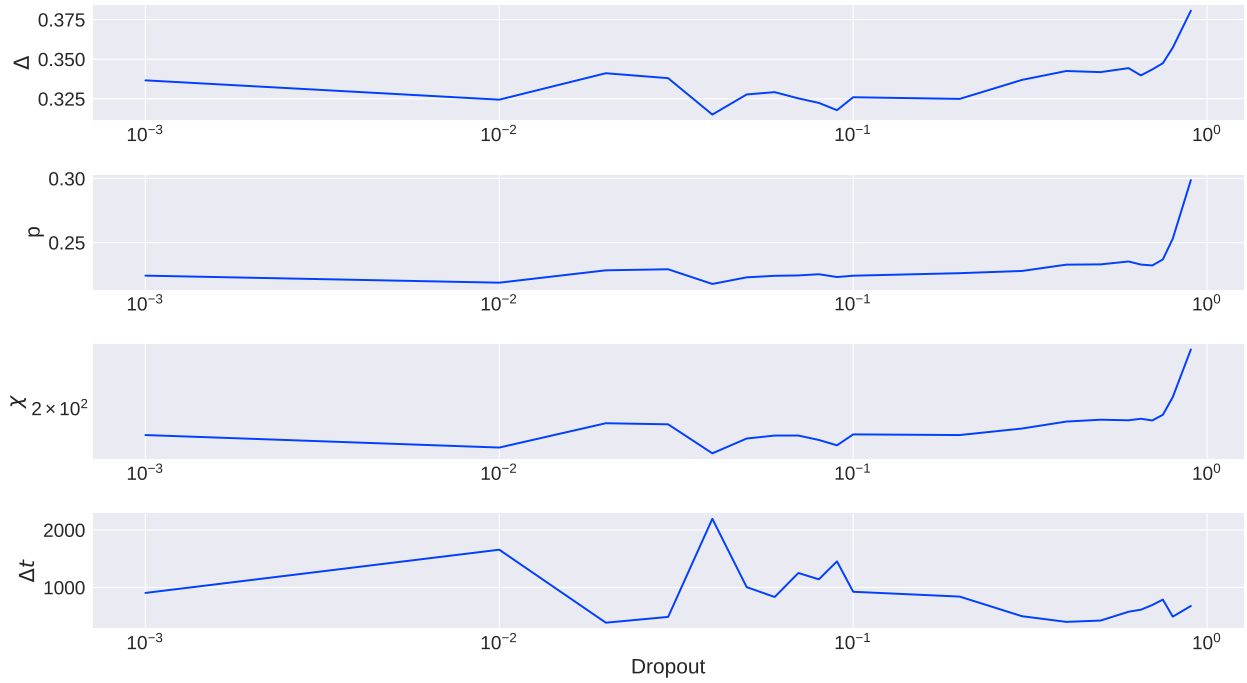

Abbildung II.4: Hyperparameteroptimierung nach Dropout nach erstem Layer

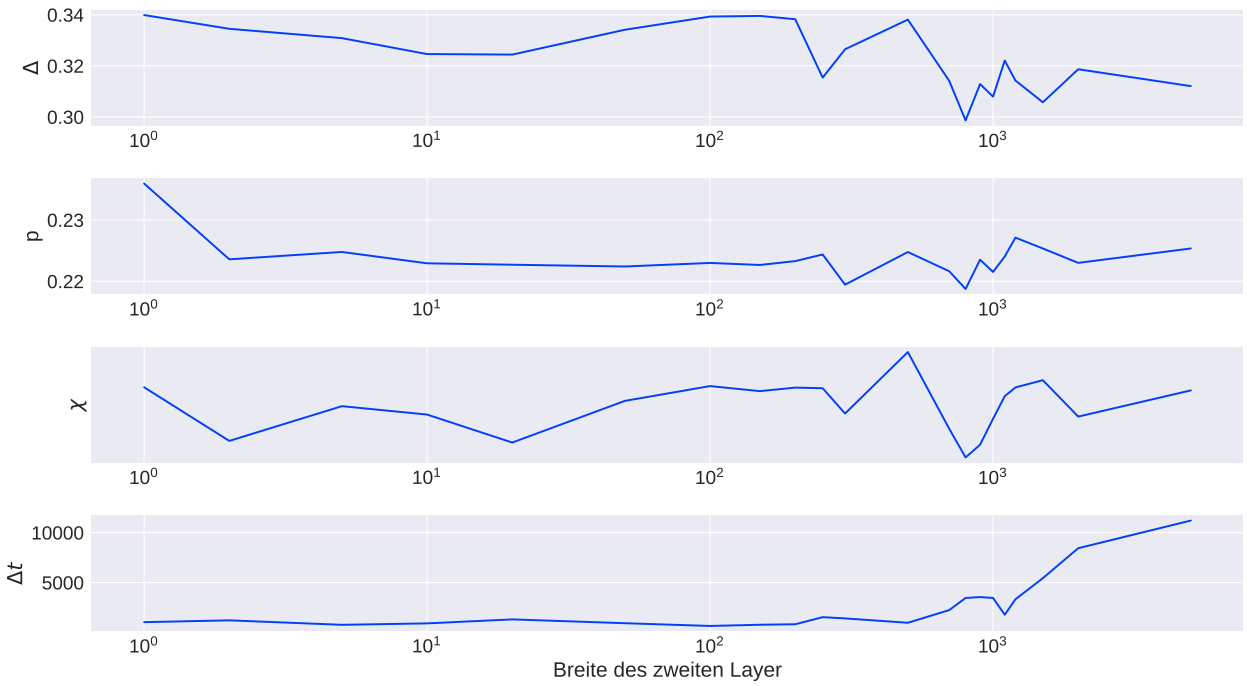

Abbildung II.5: Hyperparameteroptimierung nach Breite eines Layers (Hier des zweiten Layers)

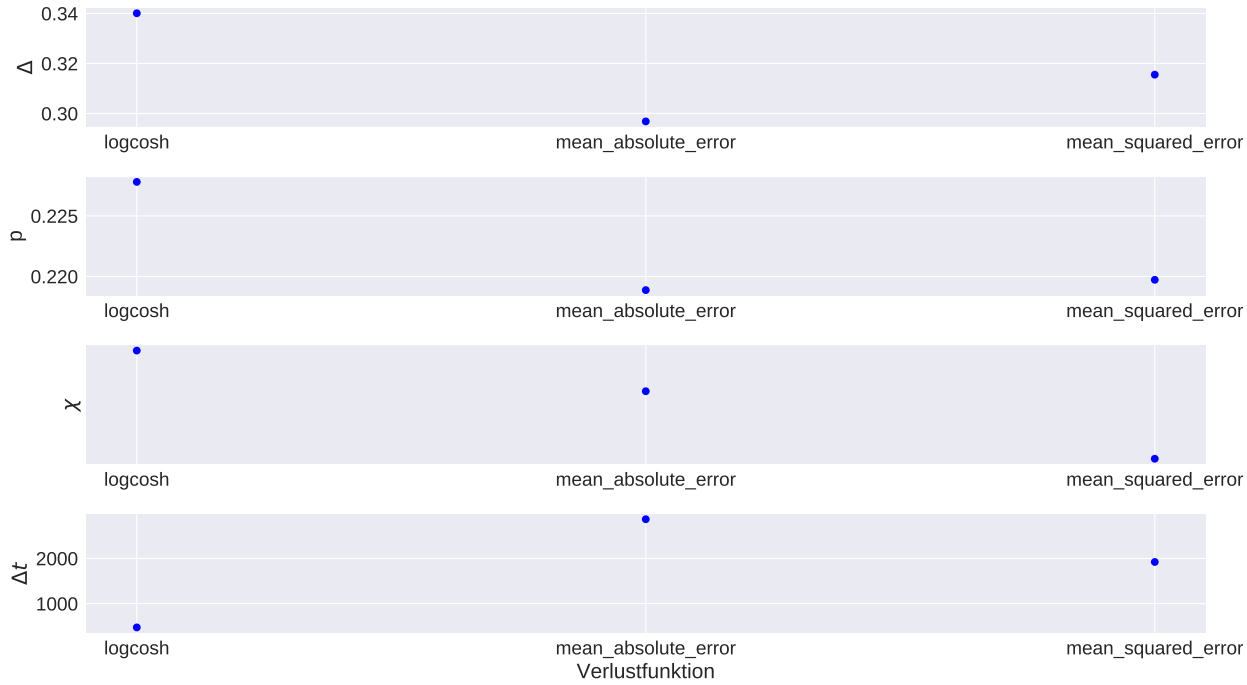

Abbildung II.6: Hyperparameteroptimierung nach Verlustfunktion

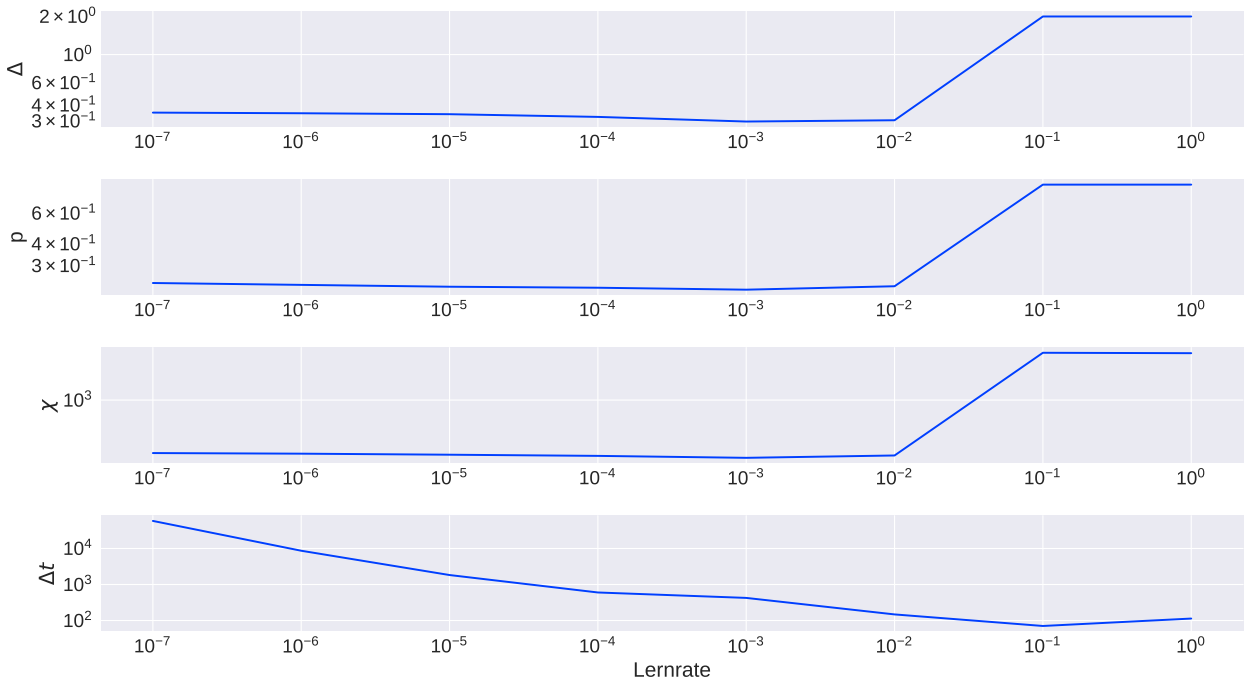

Abbildung II.7: Hyperparameteroptimierung nach Lernrate

II.2 Evolutionäre Optimierung

Die Alternative ist ein evolutionärer Ansatz: Vor jedem Training werden die Hyperparameter leicht und zufällig variiert, und diese Variation nach einem neuen Training bewertet. Ist sie besser als alles was bisher versucht wurde, definiert sie die neuen Standard Hyperparameter, ansonsten wird sie in 4 von 5 Fällen verworfen und die Standard Hyperparameter für den nächsten Schritt benutzt (in einem von 5 Fällen, werden die Hyperparameter beibehalten um fehlende Monotonie auszugleichen). Der Vorteil ist, dass monotone Optimierungsräume zwar die Entwicklung beschleunigen, aber nicht mehr benötigt werden. Der Nachteil ist, dass der evolutionäre Ansatz deutlich zu Data Leakage neigt. Nützlich ist dieser Ansatz aber trotzdem, da er gute Start Werte für die triviale Optimierung liefern kann.

III Datenoptimierung

Eine weitere wichtige Methode die Ausgabe eines Neuronales Netzwerk zu verbessern, ist es, dessen Daten so zuzuschneiden, dass diese besser Verarbeitet werden können (also beispielsweise Symmetrien auszunutzen um die Anzahl der Punkte zu senken). Die Abwägung hier ist, dass weniger Datenpunkte zwar einen kleineren Parameterraum aufspannen, und somit tendenziell besser konvergieren, aber auch weniger Informationen enthalten. Einiger der Optimierungen sind in Abbildung III.1 zu finden.

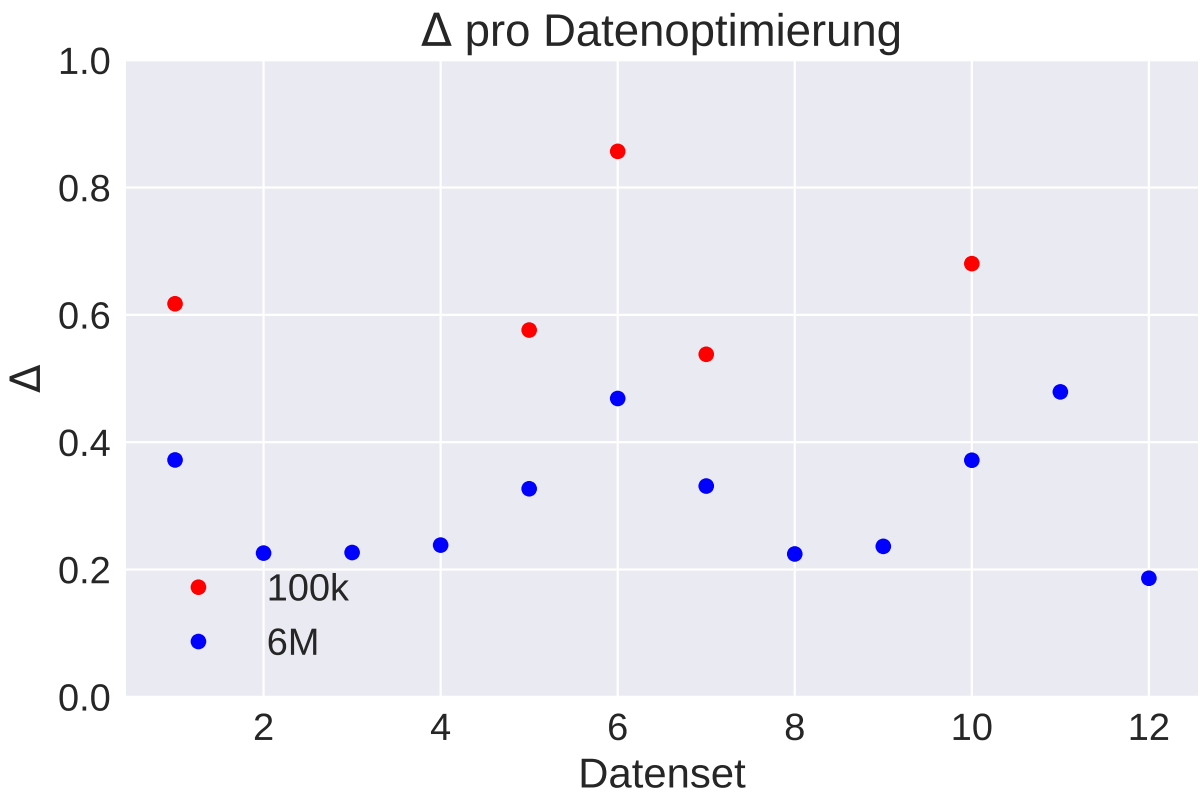

Abbildung III.1: Δ verschiedener Datensets, einmal auf 100k Trainingsdaten und einmal auf 6M Trainingsdaten, Erklärungen zu den Datensets sind in Tabelle VI.1 zu finden

Außerdem interessant ist die Korrelation zwischen dem kleinen Datenset und dem vollen Datenset, da hier auf beiden Sets trainiert wurde. Diese ist zwar nicht optimal, insbesondere

da fehlerhafte Trainingsverläufe auf dem kleinen Datenset[46], welche nicht dargestellt werden, recht häufig sind, aber man sieht doch, dass es durchaus valide ist, Parameter auf einem kleinen Datenset zu optimieren, um diese auf dem vollen Set zu benutzen.

[46]Eine fehlerhafte Verläufe ist hier eine vollständige Quantisierung, also nur die Ausgabe eines einzigen Wertes unabhängig von der Eingabe

IV Technische Details

Rohdatenfehlergeneration
Um die Fehler der Rohdaten, welche im gewichteten Mittel, sowie im Index χ benutzt werden, zu generieren, wird hier die gaussangepasste Breite der Verteilung dieser Rohdaten benutzt. Da diese für jede Ladung unterschiedlich ist, und diese natürlich auch nur für jede Ladung einzeln in etwa gaussförmig ist, sorgt diese Methode für einen Fehler für jedes Ladungsbin. Diese Fehler werden nun quadratisch kombiniert, um einen Fehler für eine Ladung zwischen zwei ganzen Zahlen zu bekommen, also $\sigma = \frac{(q_1-q)^2\cdot\sigma_1+(q-q_2)^2\cdot\sigma_2}{(q_1-q)^2+(q-q_2)^2}$. Ist die Ladung zu groß oder zu klein wird bei kleinen Abweichungen der Fehler der nächsten Ladung benutzt, während bei großen Abweichungen der Mittelwert aller Fehler benutzt wird. Dieses Vorgehen soll erreichen, dass keine Korrelationen in der Höhe der Fehler zwischen verschiedenen Eingabe impliziert werden, aber auch keine starken, normalerweise nicht vorhandenen, Schwankungen der Fehler erstellt werden.

Rigiditätsbinningauswahl
Rigiditätsbins werden in zwei Schritten ausgewählt. Zuerst werden Bins mit verschiedener Zahl von Binninggrenzen (1,2,3,5,10,20) so ausgewählt, dass für jedes Bin in etwa gleich viele Daten vorhanden sind. Für jedes dieser Datensets wird nur ein Modell trainiert (entgegengesetzt der instinktiven Erwartung geht das Training für mehr Bins schneller als für weniger Bins, da die Anzahl der Daten gesamt konstant ist, und die verbrauchte Zeit als Funktion der Eingangsdatenzahl schneller als linear steigt). Danach wird auf einem Validierungsset für jede Rigidität (natürlich in einem weiteren Binning) das optimale Netzwerk ausgewählt (das am meisten geteilte Netzwerk ist nicht unbedingt das Beste, da es weniger Daten besitzt) und die besten Netzwerke für jedes Ladungsbin neu trainiert. Schlussendlich wird nun für jedes Rigiditätsbin das Netzwerk ausgewählt, welches die besten Vorhersagen mit Ladungsteilung erreicht (dieses erneute Teilen funktioniert, da sich die zuerst gewählten Netzwerke stark überschneiden). Dieses Vorgehen soll erreichen, dass möglichst gute Netzwerke gewählt werden, während möglichst wenig Netzwerke trainiert werden müssen.

Ladungsteilung
Die Idee ein Netzwerk zu benutzen, um andere Netzwerke auszuwählen, mag zwar unlogisch klingen, da ein Neuronales Netzwerk theoretisch auch solche Strukturen darstellen kann (und zwar besser optimiert und schneller), lässt sich aber durch zwei Effekte verstehen:

- Zum einen ist der Zustandsraum weniger Ladungen bei weitem glatter, weshalb er bessere Konvergenz erlaubt
- Zum anderen neigt ein Netzwerk, welches genauer (was hier wohl äquivalent zu mehr Ladungen ist) anpassen soll, bei weitem stärker zu Overfitting, als eines welches weniger genau anpassen soll, also erlaubt das Teilen der Ladung es, den Haupteffekt, welcher die Konvergenz beschränkt zu senken

Der Preis für diese bessere Konvergenz ist allerdings, dass bei Subnetzwerken, die jeweils 3 Ladungen unterscheiden (also wie es hier gemacht wird), die Tatsache, dass Elemente welche völlig falsch vom ersten Netzwerk zugeordnet werden, praktisch zufällige Werte annehmen.

Gaußquantisierung
Der Quantisierungseffekt lässt sich so verstehen, dass ein Netzwerk nicht genügend Informationen besitzt, um zufällig streuende Ladungen vorherzusagen, es also ähnlich viele Ladungen gibt, welche größer als die reale Ladung sind, wie solche welche kleiner als diese sind, aber

trotzdem dieselbe Eingabe besitzen und schlussendlich das Modell nur nach Ladungen aufteilt und versucht für jede Ladung den minimalen Verlust zu finden, wobei der minimale Verlust einem einfachen Mittelwert entspricht. Um dieses Verständnis zu testen, wurde ein Modell trainiert, welches aus einer Zahl als Eingabe (1,2 oder 3) eine um die gesuchte Zahl gaußverteilte Zufallszahl (mit Standardabweichung 0,4, bei einer Standardabweichung von 0,8 tritt zwar auch Quanitisierung auf, aber nur ein Peak in der Mitte) vorhersagen soll. Die Ausgabe dieses Netzwerks, für zufällige (natürliche) Zahlen zwischen 0 und 5 ist in Abb. IV.1 zu sehen:

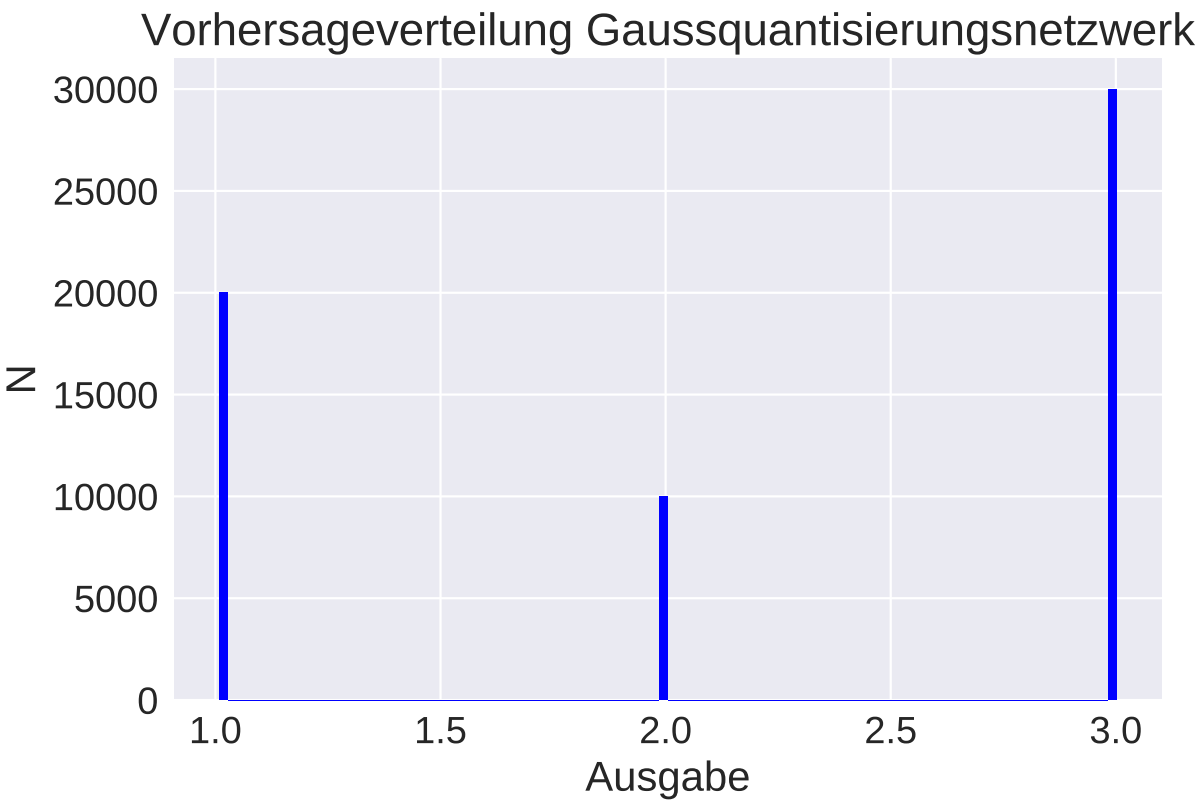

Abbildung IV.1: Ladungsverteilung Gaussquantisierung

Jeder Punkt, welcher 1 oder 0 als Eingabe hatte, wurde auf etwa 1 abgebildet, 3, 4 und 5 wurden auf etwa 3 abgebildet und 2 lediglich auf etwa 2. Hierbei ist der Mittelwert äquivalent zum Mittelwert der eintrainierten Daten, schwankt die Gaussverteilung im Training um einen anderen Wert, verschiebt sich auch die Ausgabeverteilung. Außerdem erwähnenswert ist wohl der Rundungseffekt: Ausgaben welche nicht trainiert wurden, werden auch nicht ausgegeben, sondern auf ihre nächsten Nachbarn gerundet. Dies legt nahe, dass Ladungen, welche ein Netzwerk passieren, das nicht für sie trainiert ist, werden mit hoher Wahrscheinlichkeit auf ihren nächsten Nachbarn gerundet. Dies kann ein positiver oder negativer Effekt sein, je nachdem ob man gerade Ereignisse mit Ladung 8 haben möchte, oder ob man sicherstellen möchte, dass ein Event mit einer Ladung kleiner 8 wirklich eine Ladung kleiner und nicht größer 8 hat. Glücklicherweise tritt der Rundungseffekt in dieser Form nur bei Regressionsnetzwerken auf (Klassifikationsnetzwerke nehmen keine Struktur der Ausgabewerte an, können also auch nicht runden, wobei es natürlich denselben Effekt hier auch gibt, da ein $q = 9$ Event eher aussieht wie ein $q = 8$ Event, als ein $q = 1$ Event), weshalb man das Klassifikationsnetzwerk benutzen sollte um Ladungen mit Ladung 8 zu filtern, während das Regressionsnetzwerk besser darin ist, kleine Ladungen von zu großen zu trennen.

Monte Carlo Datenvergleich
Um zu testen ob die Fehler des Netzwerks wirklich fehlerhafte Zuordnungen sind oder einfach nur fehlerhafte Vergleichspunkte besitzen, wurde das Netzwerk auf Monte Carlo Daten getestet. Dazu wurden äquivalent zu Kapitel 3.1.1 Helium Monte Carlo Datenpunkte generiert und deren Ausgabe mit der von ISS Daten mit einer angeblichen Ladung von 2 verglichen (siehe Abb. IV.2).

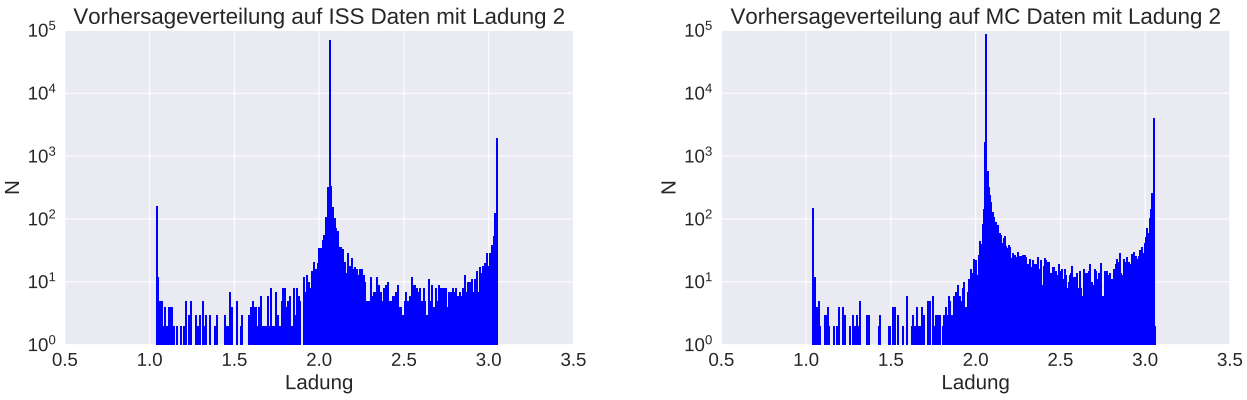

Abbildung IV.2: links: Netzwerkausgabeverteilung Helium auf ISS Daten, rechts: Netzwerkausgabeverteilung Helium auf MC Daten

Es lassen sich zwar Unterschiede sehen, insgesamt schneidet das Netzwerk auf MC Daten ein wenig schlechter ab, aber dies wiederspräche der Hypothese falscher Vergleichspunkte und lässt sich fast vollständig durch unterschiedliche Rigiditätsverteilungen beider Datensets erklären.

Eingabeladungsverteilung
Für den Verwechslungsgraphen 4.11 werden weniger nachselektierte Daten verwendet, dies bedeutet hier das zwar immer noch 2 verwendbare Ladungen benötigt werden, aber das nicht mehr auf den Abstand zur gerundeten Ladung geschnitten wird und der maximal Abstand der ToF Ladungen zur inneren Trackerladung nun 0,5 betragen darf. Die so erhaltenen Datenverteilung für Ladungen größer 2 sieht, wie in Abb. IV.3 zu sehen ist, aus:

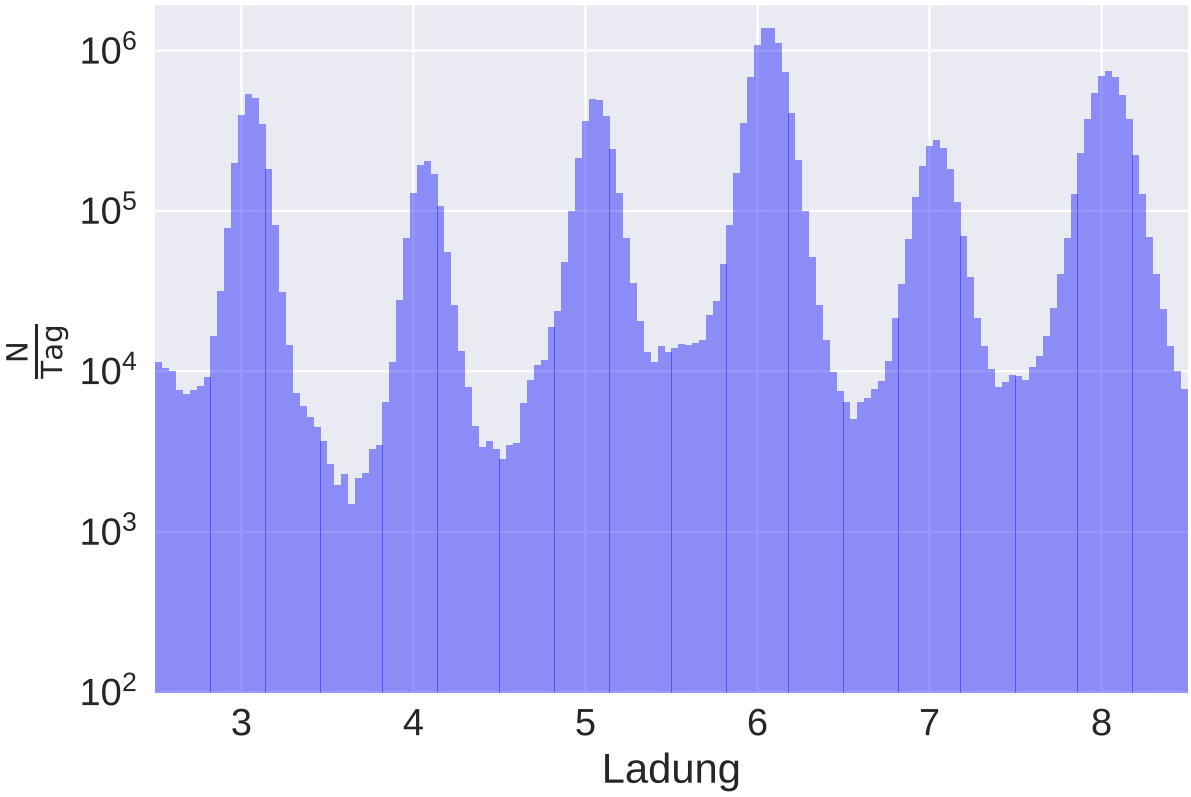

Abbildung IV.3: AMS Ladungsverteilung mit weniger stark nachselektierten Daten.

Matrixanalyse
Eine der wenigen Methoden um die Funktionsweise eines trainierten Neuronales Netzwerk besser zu verstehen, ist es sich die generierten Matrizen anzusehen, folglich hier die Matrizen des nicht geteilten Regressionsmodells. Schwarze Pixel sind verschwindende Parameter, Grüne Positive, rote Negative und die Farbsättigung gibt in einer linearen Skala die Höhe der Einträge an. In jedem Bild sind zwei Matrizen zu sehen, wobei in der ersten Matrix ist auf der X Achse die Eingabe und auf der Y Achse die Ausgabe liegt, sowie die zweite Matrix so gedreht ist, dass die Ausgabe der ersten Matrix neben der Eingabe der zweiten liegt.

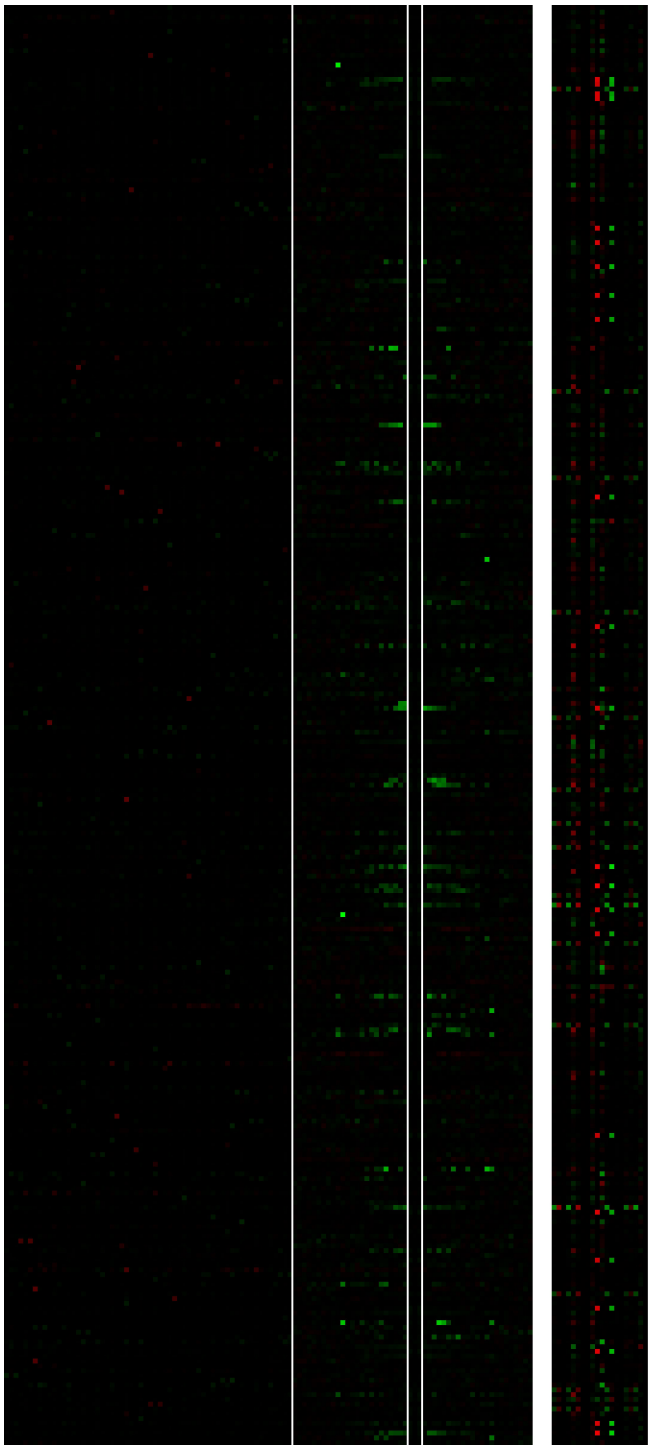

Abbildung IV.4: Regressionsmatrizen 1 und 2

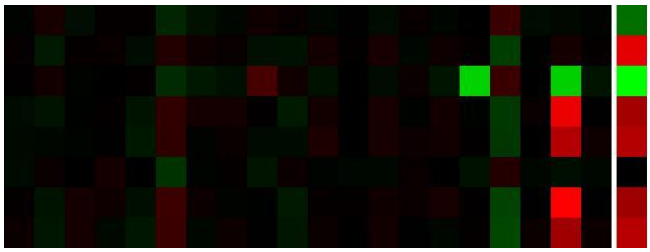

Abbildung IV.5: Regressionsmatrizen 3 und 4

Die weißen Linien in der ersten Matrix (also Abb. IV.4) sind Hilfslinien um die verschiedenen

Datenteile zu unterscheiden (vgl. Kapitel 3.1.1): Vor der ersten Linie (von links aus) sind die Daten nur rohe Eingangsdaten auf und direkt neben der Spur, nach dieser Hilfslinie folgen Summen über bestimmte Abstände, und zwischen Linie 2 und Linie 3 liegen die Summen, welche zu den rohen Eingangsdaten gehören. Man sieht wohl eindeutig, dass es mit fortschreitender Netzwerktiefe immer schwieriger wird die erkennbare Struktur der Matrix zu erkennen, aber wenn man sich die erste Matrix ansieht, lassen sich Unterschiede zwischen den beiden Eingabeteilen erkennen: Im ersten Teil steckt zwar mehr Information, aber auch weniger dicht verpackt. Folglich werden die Daten dieses Teils weit weniger stark kombiniert, was leider auch bedeutet, dass die Daten weniger gut verständlich sind. Auf der anderen Seite sieht man auch, dass die Punkte des zweiten Teils, welche zum ersten Teil gehören, im Grunde nicht benutzt werden. Das bedeutet, dass das Netzwerk theoretisch doch mehr lernen könnte, wenn man ihm rohe Daten geben würde, es nur durch Overfitting daran gehindert wird. Außerdem ist es ein gar nicht mal schlechter Index um zu zeigen, dass das Netzwerk noch nicht overfittet ist. Im zweiten Teil fallen viele verschiedene deutliche Linienstrukturen auf, welche aber jeweils paarweise unterschiedlich sind. Wenn man die verschiedenen Matrizen eines Neuronalen Netzwerks als Transformationsmatrizen sieht, welche Daten eines großen Eingangsraums auf immer kleinere Räume abbilden, bis schließlich der gesuchte Eigenschaftsraum erreicht ist, könnte man diese Linienstrukturen als Abtastfunktionen verstehen (oder mathematisch als verschiedene Kernelfunktion einer Kernelintegraltransformation), welche die Einheitsvektoren des nächsten Raums definieren. Der Vorteil dieser Interpretation ist die Tatsache, dass Punkte, welche weit weg vom Track liegen in dieser Matrix praktisch nicht angesprochen werden und somit folglich weniger wichtig für die Vorhersage sind. Dies validiert die Entscheidung nicht mehr als 50 Entfernungsbins pro Lage zu benutzen. Ähnliche Strukturen sieht man wohl auch in Matrix 2 (wenn auch weniger deutlich). Interessant ist hier, dass es Einheitsvektoren gibt, die deutlich negativ sind, und solche welche deutlich positiv sind, aber vor allem die scheinbare räumliche Separation dieser, obwohl dies keinen Effekt haben sollte. Das Vorkommen dieser Strukturen in Matrix 3 (also Abb. IV.5) zeigt wohl am ehesten die Orthogonalität dieser: Wenn man sich Eingabe 2, Eingabe 6 und die viertletzte Eingabe ansieht, also bis auf den vorletzten die drei stärksten, sieht man diese wohl eindeutig.

V Wahrscheinlichkeitsgeraden

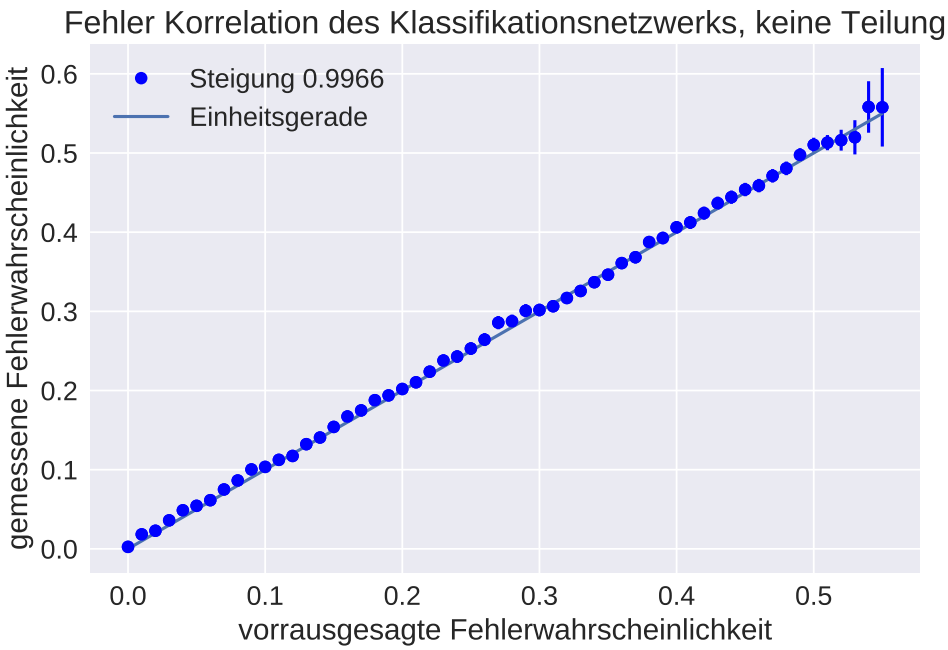

Abbildung V.1: Wahrscheinlichkeitsgerade des Klassifikationsmodells ohne Teilung

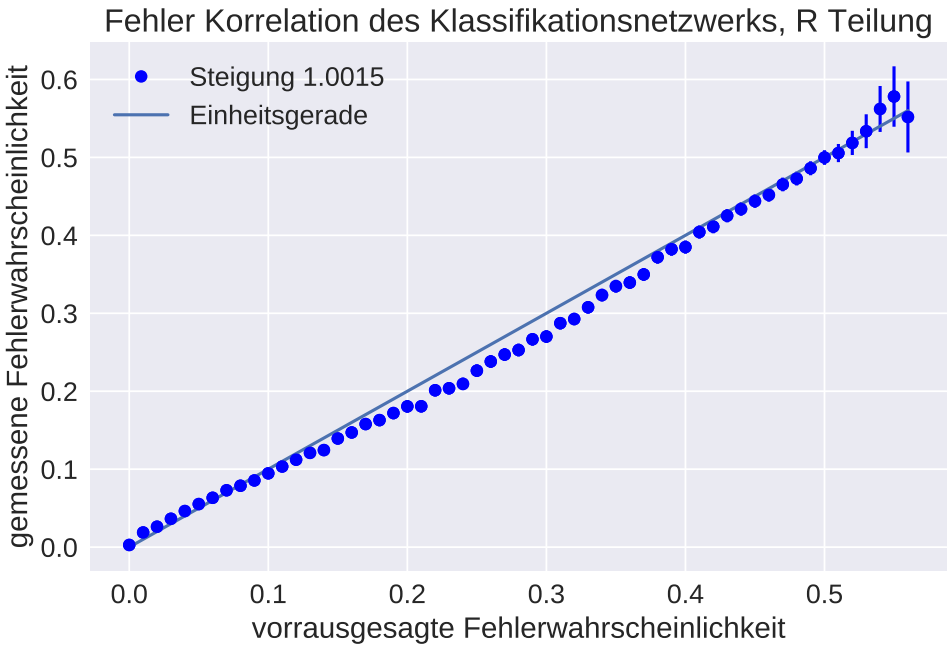

Abbildung V.2: Wahrscheinlichkeitsgerade des Klassifikationsmodells R Teilung

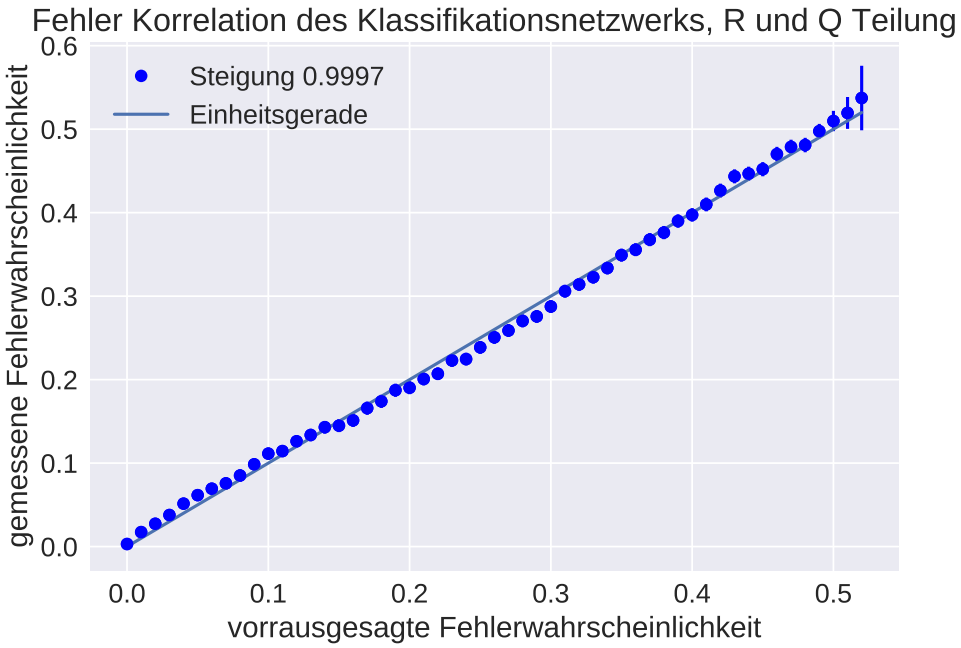

Abbildung V.3: Wahrscheinlichkeitsgerade des Klassifikationsmodells R und Q Teilung

VI Datentabellen

Datenoptimierung

Datenset	Δ 100k	Δ 6M	Beschreibung
1		0.2267	Daten auf der Spur und Summen über Spalten
3		0.2258	Daten auf der Spur und Summen über Zeilen und Spalten
4	0.8568	0.4686	wie 1, zusätzlich komplette Summe
5	0.538	0.3311	wie 9, mehr Spalten
6		0.2244	wie 1, mehr Spalten
8		0.479	nur Vorzeichen der Elemente
9	0.5761	0.3268	wie 1, nicht normiert
10		0.2364	wie 9, mit totaler Summe
11		0.1863	wie 1, trainiert auf gerundeten Werten (kleiner Wert, da sehr hohe Quantisierung)
12	0.6174	0.3721	nur die Summe aller Elemente
13	0.6804	0.3716	wie 4, nur die Summe normiert

Tabelle VI.1: Getestete Datenoptimierungsoptionen, leere Felder und fehlende Zahlen sind nicht vernünftig konvergierte Netzwerke

Interessant ist hier vor allem das normierte Datenpunkte immer deutlich bessere Ergebnisse liefern als solche, welche nicht normiert ist. Außerdem zeigt 8, welches praktisch keine Information über die Energiedeposition hat, dass diese zwar schlechter sind, aber durchaus möglich ist, auch ohne diese die Ladung eines Teilchens zu bestimmen. Außerdem wird das Netzwerk schlechter, wenn man ihm die Summe aller Werte gibt, dies lässt sich verstehen, da so ein relatives stabiles Maximum existiert, indem nur diese benutzt wird. Dementsprechend sieht man das Δ eines solchen Netzwerks (beispielsweise 13 auf 6M) nur minimal besser als das des Modells ohne Neuronale Netzwerke.

Testdatenvergleiche

Modell	Teilung	Δ val	Δ test	p val	p test
Klassifikation	keine	0,2672	0,2664	18,37%	18,27%
Klassifikation	R	0,2378	0,2374	15,37%	15,26%
Klassifikation	RQ	0,2445	0,2443	16,07%	16,08%
Regression	keine	0,2224	0,2222	16,88%	16,82%
Regression	R	0,2070	0,2068	15,05%	15,02%
Regression	RQ	0,2049	0,2050	15,05%	15,07%
Bethe Bloch		0,3966	0,3960	25,47%	25,61%

Tabelle VI.2: Testdatenvergleiche

Verwendete Tage

von	bis
22.11.2012	05.12.2012
21.02.2013	07.03.2013
26.08.2013	12.09.2013
09.04.2014	25.04.2014
13.07.2014	18.07.2014

Tabelle VI.3: gewählte Zeiten um kleine Ladungen zu generieren

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] Jörn Bleck-Neuhaus. *Elementare Teilchen*, volume 2. Springer-Verlag Berlin Heidelberg, 2013. ISBN 978-3-642-32579-3. doi:https://doi.org/10.1007/978-3-642-32579-3. Seite 40, Formel 2.14.

[2] Andrei Kounine. The Alpha Magnetic Spectrometer on the International Space Station. *International Journal of Modern Physics E*, 21(8):31, July 2012.

[3] Alpha Magnetic Spectrometer - 02 (ams-02) - 06.27.18. URL https://www.nasa.gov/mission_pages/station/research/experiments/742.html. Abgerufen am 28.08.2018.

[4] Samuel Chao Chung Ting. The alpha magnetic spectrometer on the international space station. Presentation, July 2013. URL https://ams.nasa.gov/Documents/AMS_Publications/NASA-8Jul2013.pdf. Seite 16.

[5] . NIM A 581 (2007) 156-159.

[6] Mikhail Nikolaevich Strikhanov Alexey Alexandrovich Tishchenko Alexander Petrovich Potylitsyn, Mikhail Ivanovich Ryazanov. *Diffraction Radiation from Relativistic Particles*. Springer, Berlin, Heidelberg, 2010. ISBN 978-3-642-12512-6. doi:10.1007/978-3-642-12513-3. Formel 1.72.

[7] IEKP am KIT. Homepage der ams-02 trd gruppe. URL http://www-ekp.physik.uni-karlsruhe.de/~amswww/pictures.html. Abgerufen am 18.09.2018.

[8] The trd measurement principle., . URL http://www.ams02.org/what-is-ams/tecnology/trd/. Lose nach, obgerufen am 29.08.2018.

[9] Google. Machine learning crash course, . URL https://developers.google.com/machine-learning/crash-course/. Abgerufen am 22.08.2016.

[10] initiert von François Chollet Community Projekt. Keras: The python deep learning library. URL https://keras.io/. Abgerufen am 22.08.2016.

[11] Google. Tensorflow api dokumentation, . URL https://www.tensorflow.org/api_docs/python/tf. Abgerufen am 22.08.2016.

[12] Jimmy Lei Ba Diederik P. Kingma. Adam: A Method for Stochastic Optimization. *ICLR*, 2015. URL https://arxiv.org/abs/1412.6980.

[13] Geoffrey Hinton. Rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA, 2013. URL https://www.coursera.org/lecture/neural-networks/rmsprop-divide-the-gradient-by-a-running-average-of-its-recent-magnitude-YQHki.